SYNCHRONIZATION-AVOIDING GRAPH ALGORITHMS AND RUNTIME ASPECTS

Jesun Sahariar Firoz

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Computing and Engineering
Indiana University
December 2018

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Andrew Lumsdaine, Ph.D.

Funda Ergun, Ph.D.

Predrag Radivojac, Ph.D.

Judy Qiu, Ph.D.

John Feo, Ph.D.

October 22, 2018

This thesis is dedicated to my mom, Rajvi Sultana, whose unwavering support and trust in me has been and always will be the beacon of hope for me.

Acknowledgements

The long journey of graduate studies would have been difficult if I have not met many amazing persons during this time.

First of all, I would like to thank my committee members: Andrew Lumsdaine, Funda Ergun, Predrag Radivojac, Judy Qiu, and John Feo for their time. Their valuable suggestions and unique perspectives have helped me to think differently.

Especially I would like to thank my advisor Dr. Andrew Lumsdaine for teaching me how to conduct scientific research by following proper methodologies. He always encourages new ideas. I am also thankful to him for allowing me to pursue my own research interest while providing me with timely feedback and guidance when I needed them the most. Even when I doubted myself, he always calmly shared his life experience to lift up our spirit and provided suggestions that would shed unique light to a situation. Thank you Andrew for giving me the opportunity to grow in such a nurturing environment and for encouraging curiosity.

During my affiliations with CREST (Center for Research in Extreme Scale Technologies) at IU and at PNNL, I was fortunate to collaborate with many outstanding people, who have helped me to grow as a student, as a developer, as a researcher and in general as a human being.

In particular, I would like to thank Marcin Zalewski, without whose guidance and unconditional support, this journey would not have been enjoyable. Our discussions were not only confined to research topics. We argued about the existence of God, history, financial markets, environmental issues, politics (and the motivation for being a vegetarian among others). But most of all, I learned from him what to look for in a great person and this will motivate me to try to be a good person for the rest of my life. His patience and

dedication for our betterment, both personally and academically, will always remain as an inspiration. I consider him as my mentor in every aspect of my life.

Special thanks goes to Kelsey Shephard, whom I consider as a sister who always watches over us for our well-being and goes above and beyond to support us.

I would like to thank John Feo for giving me the opportunity to spend a year at Pacific Northwest National Lab to participate in one of most interesting projects I have worked on till now. Luke D'Alessandro taught me about many standard coding practices. Martina Barnas taught me about rigorous reporting of experimental results. Ezra Kissel explained many low-level network intricacies. I am grateful to each of them for their help.

I am thankful to my lab-mates for creating an enjoyable environment for learning. Special thanks go to Timur Gilmanov, Udayanga Wickramasinghe, Abhishek Kulkarni, Daniel Kogler, and Thejaka Kanewala for their companionship.

Finally I would like to thank my mom, my aunts, my cousins, and my uncles for cheering for me from across the ocean. Mom, I am only here because of you!

Jesun Sahariar Firoz

SYNCHRONIZATION-AVOIDING GRAPH ALGORITHMS AND RUNTIME ASPECTS

Massively parallel computers provide unprecedented computing power that is only expected to grow. General-purpose Asynchronous Many-Task (AMT) runtimes exposes significant fine-grained parallelism. However, traditional Bulk-Synchronous Parallel (BSP) approach and variants thereof fail to utilize the full potential of AMT runtimes. In such execution model, parallel overheads may dominate execution time and hinder performance, especially in distributed memory settings. The synchronization overhead in particular is deeply rooted in the programming practice because it makes algorithms easier to design and implement. However, irregular applications such as graph algorithms can suffer performance bottlenecks due to the straggler effect induced by global and vertex-centric barriers. In the effort to eliminate barriers, we design and study unordered, data-driven graph algorithms, relying on optimistic (speculative) execution. Our design of algorithms allows work to be performed in any order, refining the result as the algorithm progresses. This flexibility in ordering facilitates parallel computation without global or vertex-centric synchronization. To avoid "wasted" work, our approach relies on local work prioritization.

However, performance of such algorithms is marred by two competing trends: on one hand, there is a substantial level of parallelism, which, on the other hand, necessitates runtime support with potentially high overhead and scheduling complexity. Specifically, the global work order obtained by local prioritization is susceptible to the interference from the runtime. As such, we also investigate important runtime aspects that influence the performance of graph algorithms.

In particular, we study the interaction between default runtime scheduling policies and synchronization-avoiding distributed graph algorithms. We propose plug-in scheduling policies in the application-layer for speculative graph algorithms that can provide feedback to the runtime to adjust the network progress frequency and flow of messages to the remote

destinations. These techniques are useful for unordered distributed graph algorithms that necessitate a balanced interleaving of communication and computation to achieve better performance.

Graph algorithms designed in different programming models have vastly different workload characteristics. In the final part of the thesis, we propose adaptivity of the runtime parameters such as message coalescing and flow control to adjust the "pressure points" in the runtime due to variable workload characteristics over the execution of an algorithm.

Andrew Lumsdaine, Ph.D.

Predrag Radivojac, Ph.D.

Funda Ergun, Ph.D.

Judy Qiu, Ph.D.

John Feo, Ph.D.

Contents

List of Figures

List of Acronyms

CHAPTER 1

Introduction

Graphs and graph algorithms have long been an important part of fundamental computer science. Recently, graphs have received significant attention because of their utility to data analytics. As data analytics problems have exploded in size and complexity, the need for scalable graph algorithms has exploded along with them.

Although parallel graph computations have unique challenges [78], parallelization of graph algorithms is still typically accomplished using paradigms established long before the need for scalable graph algorithms emerged. Although these approaches (Communicating Sequential Process (CSP) , Bulk Synchronous Parallel (BSP) , Single Program, Multiple Data (SPMD) , et al) have proven to be highly effective for scientific computing and numerical linear algebra, they have proven to be much less effective for graph algorithms. Notably, relatively coarse compute-communicate phases are poorly matched to the fine-grained, irregular computation and communication structures exhibited by graph problems.

Moreover, graph algorithms, as described in the literature and in textbooks, are often developed to optimally solve certain problems in a theoretically ideal setting. The subsequent prescription of data dependencies and ordering of operations leads to severe synchronization when these algorithms are parallelized. Some success has been achieved by relaxing synchronization (e.g., with Δ-stepping for solving SSSP problems [85]), yet even with relaxed ordering [64], the scalability of these algorithms is still ultimately limited by synchronization. In limited scale, asynchrony has been shown to be applicable for monotonically increasing or decreasing updates [114] that requires periodic global synchronization for termination detection.

Synchronization in graph computation is pernicious. Global synchronization barriers imposed by BSP approaches introduce the *straggler effect* [64] where the whole distributed system must wait for the last straggler to move to the next step. The larger the system gets, the more pronounced the effect.

Additionally, we identify another kind of barrier encoded in certain graph algorithms, which we call *vertex-centric barriers*. This type of synchronization arises when an ordering is imposed on vertices before starting an algorithm execution, resulting in an implicit predecessor-successor relationship among vertices and all their neighbors, thus forming a *pre-execution Directed Acyclic Graph* (DAG). Many graph algorithms depend on such vertex ordering to determine which vertex to process first.

For example, the problem of finding an optimal coloring of a graph is NP-complete [53] . However, over the course of time, many heuristic parallel *greedy* algorithms, based on Luby's [77] iterative maximal independent set computation, have been devised that perform well in practice. To make a trade-off between execution time and coloring quality, greedy coloring algorithms apply different vertex ordering criteria when assigning priorities to vertices so as to decide which vertex to color first. In doing so, an implicit predecessor-successor relationship between a vertex and its neighbors is formed. This can be visualized as a directed acyclic graph (DAG), termed as *priority DAG* [18], with edges emanating from the predecessor(s) to the successor(s). Once the implicit DAG is created, most algorithms proceed in steps and traverse the DAG in a *DAG-synchronous* fashion: each vertex in a sub-DAG waits until all its predecessors are colored and then color itself with an available color not taken by any of its predecessors. At the level of a single vertex, this resembles Bulk-synchronous-parallel (BSP) execution model, where an algorithm iterates through computation, communication, and synchronization steps.

In greedy coloring algorithms, waiting on a predecessor gives rise to **vertex-centric barrier/synchronization** (Fig. 1.1). Vertex-centric barriers induce similar ramification as global synchronization barriers in BSP approach, where the whole system must wait for a straggler before moving to the next step. A vertex with a large number of predecessors

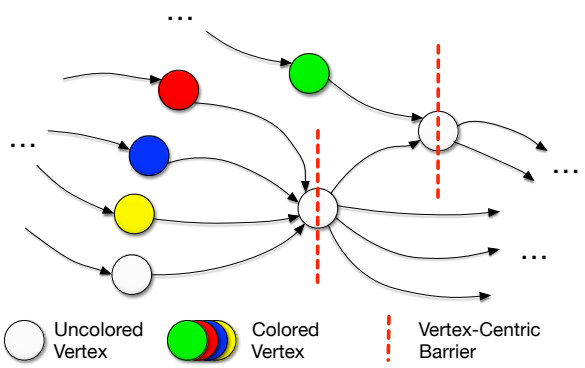


FIGURE 1.1. Vertex-centric barriers.

(straggler) impedes other vertices in the same level from advancing (straggler effect). Although this may not demonstrate itself as a problem in a shared-memory implementation, straggler effect can seriously limit performance of an algorithm in a distributed setting.

To alleviate the problem of straggler effect associated with global synchronization and vertex-centric barriers, we consider the class of *unordered* algorithms with large available parallelism [63] that can be executed *optimistically* (speculatively) [71]. Unordered algorithms allow tasks to be performed in any order and correctness is still guaranteed. This flexibility in ordering facilitates parallel computation without global or vertex-centric synchronization. However, too much speculation causes explosion of sub-optimal work. If results calculated in intermediate steps are sub-optimal and require updates too often, then performance suffers. Our objective is to formulate an algorithmic approach based on unordered execution to design synchronization-avoiding graph algorithms that demonstrate better performance while retaining good scalability.

Traditional BSP-like approach helps to execute an algorithm with optimal amount of work and performs least amount of useless work. While minimizing useless work is crucial in a sequential computation, it is the opposite of creating parallelism. Relaxing the constraints of work optimality, a graph can be explored optimistically correcting results as better information becomes available (giving rise to *label-correcting* algorithms). In avoiding synchronization, our approach relies on speculative (or optimistic) execution. As a consequence, synchronization-avoiding algorithms are necessarily *label-correcting*.

Our approach leverages an important observation: label-correcting algorithms can be classified according to whether or not the vertex property update (the label correction) is monotonic or non-monotonic. This classification can in turn be identified with the subsequent synchronization behavior of the algorithms in each category. Algorithms with monotonic updates (such as algorithms for solving single-source shortest paths or connected components) exhibit global synchronization, whereas non-monotonic algorithms (such as algorithms for vertex coloring) exhibit vertex-centric synchronization.

To support effective asynchrony in the communication framework, we employ *active messages* [105], where messages are sent explicitly but receives are implicit. Here, since a pre-registered receive handler knows a priori the address of user-level handler, it can directly extract the computation from a message and integrate it to the ongoing computation. There is no need for the endpoints to get involved as well as no collectives (gather, scatter etc.) are needed. In doing so, active messages (AM) eliminate software overhead of buffering and allow for more overlapping between communication and computation and expose more asynchrony.

1.1. Techniques For Synchronization-avoiding Graph Algorithms

We have developed algorithms that avoid global synchronization and vertex-centric barriers. To do so, our approach relies on the following supporting mechanisms:

(1) Adapting graph applications to execute graph operations optimistically (speculatively) in any order without affecting the correctness of the final result. These *unordered* algorithms refine the results as the algorithms progress (hence label-correcting) and produce the correct result at the end,

(2) Expressing graph operations in terms of fine-grained active messages [105] and handlers with extended capabilities to send computations to the targets asynchronously (thus spawning asynchronous tasks at the targets),

(3) Relying only on local atomic operations to update properties,

(4) Exploiting the ability to invoke unbounded-depth active message handlers to chain computations,

(5) Guiding the computations with thread-local priority-metric(s) based ordering to navigate through a better execution trace,

(6) Applying runtime optimizations such as message coalescing, reduction and routing transparently, and

(7) Eliminating the need to buffer messages for computations with non-monotonic updates, so that the complete asynchronous execution does not increase memory requirement for buffering out-of-band messages (for example graph coloring).

Combining these ideas have several advantages. For example, unordered algorithms unveil maximum asynchrony and parallelism due to the independence of task execution. Asynchronous spawning of tasks with active messages and relying only on atomic operations for vertex property updates avoid overheads associated with global and vertex-centric synchronization barriers and sophisticated (and expensive) vertex locking protocols. Functionally-flexible unbounded-depth active-message handlers propagate the computation forward, rather than restricting themselves only in replying to the senders of the active messages. Additionally, due to the nature of graph structure, variable-length computation strands arise during the execution of a graph algorithm. Such computations can be chained easily with extended active messages.

Moreover, the active-message handlers can assist the runtime to decide whether to continue with application-level computation that can trigger more sends or whether to transfer control to the runtime for progressing the runtime (termed as *flow control*). Thread-local ordering prevents work explosion by minimizing execution of sub-optimal work while keeping the ordering overhead to a minimum by avoiding contentions on priority queue data-structures. Finally, runtime optimizations can be applied transparently: message reduction (caching) can eliminate redundant messages; message coalescing can bundle messages targeted for a particular destination to utilize the network bandwidth properly.

1.2. Context Matters: Distributed Graph Algorithms and Runtime Systems

The increasing complexity of the software/hardware stack of modern supercomputers makes understanding the performance of the modern massive-scale codes difficult. Distributed graph algorithms (DGAs) are at the forefront of that complexity, pushing the envelope with their massive irregularity and data dependency. We analyze the existing body of research on DGAs to assess how technical contributions are linked to experimental performance results in the field. We distinguish *algorithm-level* contributions related to graph problems from *runtime-level* concerns related to communication, scheduling, and other low-level features necessary to make distributed algorithms work. We show that the runtime is an integral part of DGAs' experimental results, but it is often ignored by the authors in favor of algorithm-level contributions. We argue that a DGA can only be fully understood as a combination of these two aspects and that detailed reporting of runtime details must become an integral part of scientific standard in the field if results are to be truly understandable and interpretable. Based on our analysis of the field, we provide a template for reporting the runtime details of DGA results, and we further motivate the importance of these details by discussing in detail how seemingly minor runtime changes can make or break a DGA.

1.3. Runtime Scheduling Policies for Synchronization-avoiding Graph Algorithms

The performance of a distributed graph algorithm is deeply tied to the context of the underlying distributed software/hardware stack, here collectively referred to as the runtime. This thesis additionally explores the scheduling and runtime system support for executing unordered graph computations, where there is a substantial level of parallelism to exploit but one that also necessitates runtime support with potentially high overhead and scheduling complexity, such as optimistic (speculative) execution. The flexibility of ordering in unordered algorithms facilitates parallel computation without global or vertex-centric synchronization. To avoid "wasted" work, our approach relies on local work prioritization. However, the global work order obtained by local prioritization

is susceptible to the interference from the runtime. Such interference can delay timely delivery of messages containing better information. As a consequence, with sub-optimal work ordering, more time will be spent on correcting the speculative computation than on useful work.

Striking the balance between speculative computation and progressing the network to propagate better messages has to be done at the level of the graph algorithm, however communication and scheduling are aspects of the runtime. To bridge that gap, graph algorithms need runtime hooks to influence scheduling policies. To get better performance, the common case requires algorithm-specific and graph-type-specific scheduling policies. To provide appropriate scheduling on application-specific basis, application needs a plug-in mechanism to provide the best scheduling policies to the runtime.

To this end, it is imperative to distinguish runtime-level *tasks* from application-level *work items* and to make informed decision about executing tasks or executing work items at any particular instance of time. Runtime-level tasks encompass scheduling lightweight threads, probing and progressing the underlying transport, and detecting termination. Application-level work items are generated by the application (for example relaxing a vertex distance). While executing an algorithm, a general-purpose runtime stack is responsible for timely delivery of work items from lower-level bit transport to the application level. Work items may pass through several lower-level buffers (runtime specific) to exit the runtime world and then be handed over to the application. For applications that are sensitive to work item delivery characteristics, the runtime must cooperate with the application to deliver optimal performance by reducing sub-optimal work.

To find a way to reduce semi-optimal work for unordered graph algorithms, we ask the question: are the scheduling policies encoded into the default runtime scheduler good enough to schedule tasks and work items for supporting optimistic parallelization? If not, can we improve scheduling of the application tasks by utilizing feedback from the application?

In this connection, we propose the concept of plug-in scheduling policies that augment the scheduler of the underlying runtime to adapt it to a specific application. We present several implementations of our approach in an asynchronous many-task runtime system (AMT), and we demonstrate that the implementation using a plug-in scheduling policy is the most efficient, outperforming other versions and even the well-known Δ-stepping algorithm. We achieve the performance using two heuristics, *flow control* and *adaptive frequency of network progress*, providing the evidence that adequate runtime support for irregular distributed algorithms is vital for their performance.

Our contributions are to demonstrate combining an unordered computation with AMTs and to demonstrate the benefits of lifting and delegating some responsibilities to the domain experts by providing a plug-in scheduler in the underlying runtime.

1.4. Adaptive Runtime features for Distributed Graph Algorithms

Performance of distributed graph algorithms can benefit greatly by forming rapport between algorithmic abstraction and the underlying runtime system that is responsible for scheduling work and exchanging messages. However, due to their dynamic and irregular nature of computation, distributed graph algorithms written in different programming models impose varying degree of workload pressure on the runtime. To cope with such vastly different workload characteristics, a runtime has to make several trade-offs. One such trade-off arises, for example, when the runtime scheduler has to choose among alternatives such as whether to execute algorithmic work, or progress the network by probing network buffers, or throttle sending messages (termed flow control). This trade-off decides between optimizing the throughput of a runtime scheduler by increasing the rate of execution of algorithmic work, and reducing the latency of the network messages. Another trade-off exists when a decision has to be made about when to send aggregated messages in buffers (message coalescing). This decision chooses between trading off latency for network bandwidth and vice versa. At any instant, such trade-offs emphasize either on improving the quantity of work being executed (by maximizing the scheduler

throughput) or on improving the quality of work (by prioritizing better work). However, encoding static policies for different runtime features (such as flow control, coalescing) can prevent graph algorithms from achieving their full potential, thus can undermine the actual performance of a distributed graph algorithm . In this thesis, we also investigate runtime support for distributed graph algorithms in the context of two paradigms: variants of well-known Bulk-Synchronous Parallel model and asynchronous programming model. We explore generic runtime features such as message coalescing (aggregation) and flow control and show that execution policies of these features need to be adjusted over time to make a positive impact on the execution time of a distributed graph algorithm. Since synchronous and asynchronous graph algorithms have different workload characteristics, not all of such runtime features may be good candidates for adaptation. Each of these algorithmic paradigms may require different set of features to be adapted over time. We demonstrate which set of feature(s) can be useful in each case to achieve the right balance of work in the runtime layer. Existing implementation of different graph algorithms can benefit from adapting dynamic policies in the underlying runtime.

1.5. Contributions

Our contributions in this thesis are as follows:

- A new approach to developing scalable graph algorithms, based on eliminating global synchronization and vertex-centric barriers, and distributed termination detection that relies on four-counter based Sinha-kale-Ramkumar algorithm [103] (Chapter 3).
- Identification of two classes of algorithms (monotonic and non-monotonic) corresponding to two classes of synchronization (global and vertex-centric) (Chapter 3).
- Application of our approach to develop new algorithms for solving SSSP, connected-components, and vertex coloring. The developed algorithms incorporate the seven principles discussed in Sec. 1.1 and are implemented using the Active Pebbles model [110] (designed for fine-grained data-driven irregular

applications). We show that our algorithms outperform several baseline algorithms. Comparison of our algorithms with algorithms from the well-known Powergraph [55] framework shows significantly improved scalability. We also compare the performance of our algorithms with Gemini [115] distributed graph framework and Galois [91] shared-memory graph framework (Chapter 3).

- We show that runtime considerations are inseparable from algorithmic concerns in performance engineering of large-scale distributed graph algorithms. We identify, classify, and discuss two levels of distributed graph algorithms:
  - Application-level aspects that authors identify as the main algorithmic contributions of their research.
  - Runtime-level aspects that authors do not explicitly consider a part of the algorithm but that play a crucial role in the overall performance.

  We argue that the whole system stack, starting with the algorithm at the top down to low-level communication libraries must be considered. With a set of carefully designed experiments of runtime features, we show how runtime can make or break an algorithm (Chapter 4).
- We demonstrate the benefits of lifting and delegating scheduling responsibilities from the runtime level to the domain experts in the application level by providing a plug-in scheduler in the underlying runtime. With such minimal plug-in scheduling interface to the runtime, application programmer can provide effective feedback to the underlying runtime to achieve better performance for unordered algorithms (Chapter 5).
- We show that, based on the hints available in the application level, flow control and network progress frequency in the runtime can be regulated from the application level with the help of a plug-in scheduler to improve the execution profile of our algorithmic approach (Chapter 5).
- We identify a set of pressure-points in the lower stack of graph applications aka runtime. Based on graph algorithms' characteristics, we demonstrate how

adapting dynamic policies to adjust these pressure-points can benefit such graph algorithms (Chapter 6).

- We demonstrate that adapting dynamic message aggregation policy can speedup graph algorithms that execute in a (relaxed) level-synchronous fashion (for example, Δ-stepping [85] and K-level asynchronous (KLA) [60] single-source shortest paths algorithms) (Chapter 6).
- We show how dynamically adapting runtime-level flow control mechanism can boost performance of asynchronous graph algorithms that are based on optimistic parallelization (Chapter 6).

1.6. Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 discusses the baseline algorithms we consider for single-source shortest paths, connected components and graph (vertex) coloring problems. We also give a brief overview of the active pebbles model and how active pebbles model can be leveraged to design synchronization-avoiding graph algorithms. In addition, we discuss ordering of tasks at different levels in distributed graph algorithms. Chapter 3 describes in details our approach for designing synchronization-avoiding graph algorithms. We demonstrate that our algorithms outperform well-known baseline algorithms discussed in Chapter 2 as well as the algorithms in the PowerGraph framework for several graph problems at larger scale. Chapter 4 presents evidence of impact of low-level transport, scheduler, and hardware, which we refer to as the *runtime*, on large-scale distributed graph algorithms. To strike a balance between communication and computation, our approach may require proper runtime support. Such support can be provided either as plug-in scheduling policies in the application layer or adapting runtime features on-the-fly in the runtime layer according to the workload. Chapter 5 focuses on the impact the runtime system - and, specifically, its scheduling policies - have on distributed-memory graph processing algorithms which avoid synchronization overheads. There is a basic tension that arises between driving the progress of asynchronous

communication (task) and performing the work of the application (work item). To reduce such tension, proper scheduling policies are necessary for our synchronization-avoiding graph algorithms for better performance. We discuss in details how such policies can be enforced through plug-in schedulers. In addition, graph algorithms designed in different programming models have changing workload characteristics. A runtime has to consider trade-offs between optimizing throughput and minimizing message latency depending on the algorithm. To handle the varying workload, we propose adaptivity of different runtime features in Chapter 6. Chapter 7 discusses future directions for research. We summarize the contributions of this work and provide concluding remarks in Chapter 8 .

CHAPTER 2

Background

We denote an undirected graph with n vertices and m edges by $G(V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$ represent vertex set and edge set respectively. Each edge $e_i \in E$ is an unordered pair $\{v_j, v_k\}, v_j, v_k \in V$. In this thesis, we evaluate our approach for three graph kernels: single-source-shortest paths, connected components and graph coloring. In this chapter, we briefly discuss the baseline algorithms we implemented to compare the performance of our algorithms with. We choose these algorithms as baselines for comparison because these algorithms are formulated based on the observations on input graph structures and topologies. Although some variants of these algorithms, specially for SSSP, has been proposed recently [81, 26], these variants heavily depend on graph pre-processing step [81] or system-specific optimizations (such as SPI library on BlueGene-Q machine in [26]). In addition, this chapter gives a brief overview of the Active Pebbles model and how this model supports asynchronous graph execution. Asynchronous execution of graph algorithms may result in the execution of sub-optimal work. We also discuss in this chapter how ordering in different spatial level can reduce sub-optimal work execution and can help in avoiding work explosion.

Hassan et al. [98] classify algorithms into two main categories of *ordered* and *unordered* algorithms. Ordered algorithms require ordering of tasks for correctness whereas unordered algorithms do not depend on any order of tasks. Parallelizing ordered algorithms is challenging as the parallel execution must still maintain the ordering. On the other hand, unordered algorithms are easier to parallelize as tasks can be executed in any order. Furthermore, both of the kinds of algorithms may, in addition, rely on *task priority*, where executing a task with the highest priority is either required (ordered algorithms) or beneficial (unordered algorithms).

2.1. SSSP Algorithms

Assume that each edge in graph G has an associated cost (weight) > 0. In *single-source shortest path (SSSP)* problem, given a graph G and a source vertex s, we are interested in finding the paths with shortest weight (distance) between s and all other vertices in the graph. The traditional greedy Dijkstra algorithm [39] employs a priority queue to always process one of the vertices with the best distance, ensuring work optimality. It is a "label-setting" algorithm, where the computed vertex distance is final. The algorithm is an ordered algorithm, and it must execute the tasks in the order dictated by their distances where a smaller distance is better. Dijkstra's algorithm is *work optimal*, performing the least amount of *useless work* that does not update the distance of a vertex. Distributed Dijkstra's algorithm requires a global priority queue to maintain the order, and even with Crauser's et al. [36] improvements, the order imposed by the queue is inherently sequential.

While Dijkstra's algorithm minimizes the amount of useless work, it is also the opposite of creating parallelism. Relaxing the constraints of work optimality, a set of active vertices in a graph can be explored speculatively and in parallel, correcting results as better information becomes available (giving rise to the *label-correcting* algorithms). In the basic label-correcting SSSP algorithm, a task is a vertex-distance (v, d) pair that, when executed, updates the distance of the vertex v to the distance d if d is better than the current distance d_v for v. Such unordered algorithms can perform tasks in any order, but the further the task execution order strays from the Dijkstra's order, the more work is performed that then needs to be *invalidated* by updating the distances it produced with better new distances. Thus, the better the unordered algorithm approximates Dijkstra's order, the less work it performs.

2.1.1. Δ-stepping Algorithm. The Δ-stepping [85] algorithm approximates the ideal priority ordering by arranging tasks (vertex-distance pairs) into distance ranges (buckets) of size Δ and executing buckets in order. Within a bucket, tasks are not ordered, and can be executed in parallel. After processing each bucket, all processes must synchronize

before processing the next bucket to maintain task ordering approximation. The more buckets (the smaller the Δ value), the more time spent on synchronization. Similarly, the fewer buckets (the larger the Δ value), the more sub-optimal work the algorithm generates because larger buckets provide less ordering.

Δ-stepping SSSP algorithm is described in Alg. 1. The algorithm starts by putting source s with distance 0 into Bucket B_0. In each epoch (superstep) i, vertices within the distance range $i\Delta - (i+1)\Delta$ from the source contained in a bucket B_i are processed in parallel by worker threads (Ln. 12). If a vertex distance is updated, the updated distance is propagated to all of its neighbors (Ln. 16). Processing a bucket may produce extra work (vertex-distance pair) for the same bucket or for the successive buckets. Worker threads cannot proceed to the next bucket unless all workers on each of the distributed node have finished processing vertices contained in the current bucket. This requires global synchronization barrier (which is implicit in the epoch). The message handler $explore$ is responsible for putting a vertex in the appropriate bucket if the updated distance is better (Line 10).

2.1.2. k-level Asynchronous (KLA) Algorithm. In k-level asynchronous (KLA) algorithm [60], within each superstep i, computation can proceed optimistically for vertices that are within the range of $[(i-1)k, ik]$ levels from the root. Vertices that are reachable within these levels in a superstep can be processed in parallel. However, a global barrier is required after each superstep. The algorithm is shown in Alg. 2. The algorithm starts with two buckets, containing vertices within the range of $((i-1)k, ik]$ and $(ik, (i+1)k]$ to be processed in the current superstep and the next superstep respectively. The algorithm starts by processing vertices in the current bucket(Ln. 13) and sending the updates to its neighbors if a better distance has been found (Ln. 17). Once the update has been received, the algorithm decides whether this vertex will be processed in the current superstep or the next (Lns. 6–7) and put it in the appropriate bucket (Ln. 8).

KLA algorithm (Fig. 2.1) is based on graph topology (number of vertices in k levels that can be processed in parallel) whereas Δ-stepping algorithm (Fig. 2.2) is based on graph

Alg. 1: Δ-stepping Algorithm

In : Graph $\mathcal{G} = \langle V, E \rangle$, source s, Parameter Δ

$\forall\, v \in V$: $owner[v]$ = rank that owns v

Out: $\forall\, v \in V$: $distance[v]$ = distance of v

```
1  distance[v] ← ∞, ∀v ∈ V;
2  B ← set of buckets based on Δ;
3  i ← 0;
4  enqueue (s, 0) ← B_i;
5  message handler explore(Vertex w, distance d)
6      if d < distance[w] then
7          oldindex ← D(w)/Δ;
8          B_oldindex ← B_oldindex \ w;
9          newindex ← d/Δ;
10         B_newindex ← B_newindex ∪ w enqueue w → B;
11 while B not empty do
12     while B_i not empty do
13         active message epoch
14             parallel foreach v ∈ B_i do
15                 if Update distance[v] then
16                     parallel foreach neighbor w of v do
17                         d = distance[v] + weight(v, w);
18                         send explore(w, d) to owner(w);
19     i ← i + 1;
```

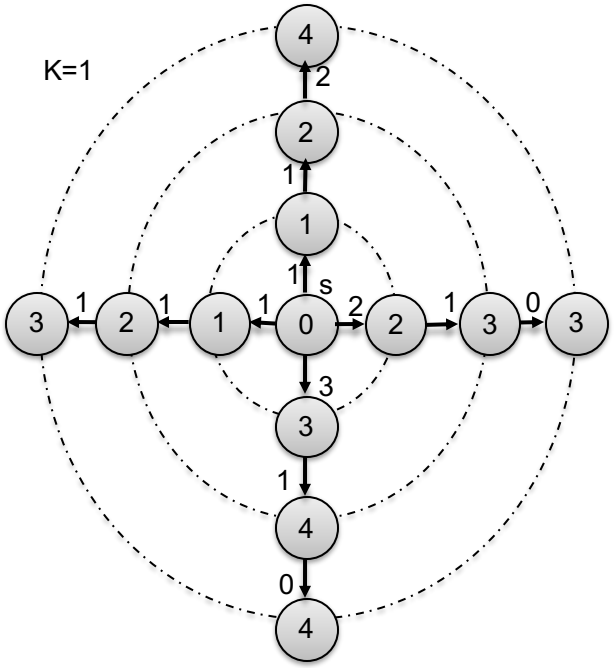


FIGURE 2.1. K-level asynchronous algorithm.

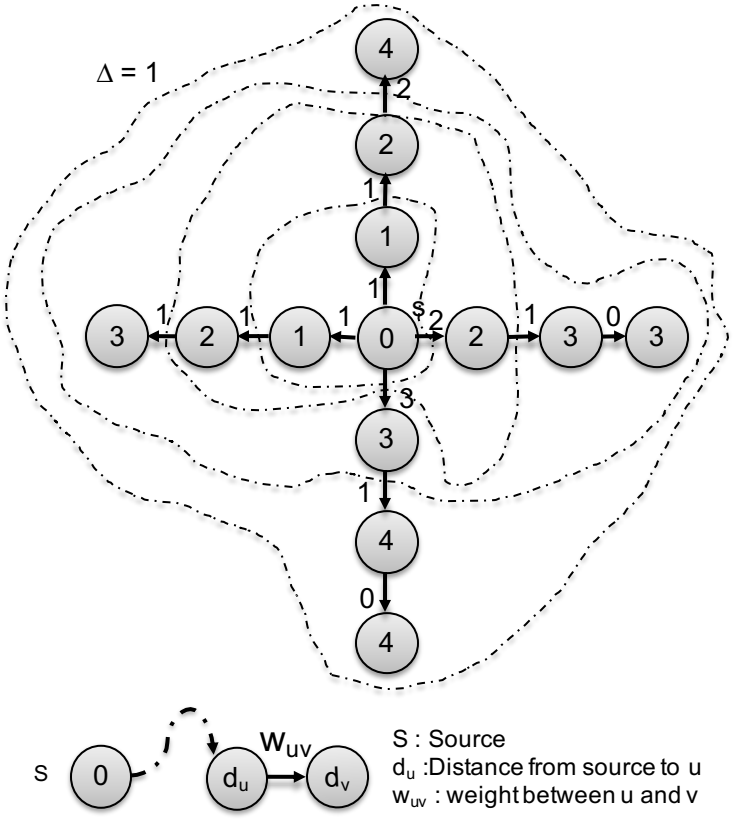


FIGURE 2.2. Δ-stepping algorithm .

property (number of vertices that can be processed in parallel within current Δ distance range). Both algorithms require global synchronization.

2.2. Connected Component Algorithms

A connected component (CC) of a graph G is a maximal subgraph of G to contain a path between every pair of vertices in the subgraph. An algorithm for finding connected components discovers the component membership of each vertex in a graph. In practice, two approaches are used for CC, namely *hooking-shortcutting* and label-propagation. Shiloach-Vishkin (SV) algorithm [102] for finding CCs is based on hooking-shortcutting. Initially each vertex belongs to its own individual component. In hooking phase , trees are hooked together if there exists an edge between them. Shortcutting flattens the trees by hooking them to one root per component. The algorithm requires global synchronization after each hooking and shortcutting phases. Recently, a variant of SV algorithm has been proposed in [65] where, at the beginning of execution, a parallel Breadth-first search (BFS) is executed from high-degree vertices to discover the largest component in the graph and then proceed with traditional SV algorithm for the rest of the computation. This algorithm has been shown to outperform multi-step algorithm proposed in [104]. We also implement BFS-based SV algorithm and call it parallel BFS (PBFS). All these algorithms require global synchronization.

Label propagation algorithms starts by initializing each vertex with an unique ID and propagates vertex labels based on minimum (maximum) label. Our algorithm differs from these algorithms as we prioritize labels during algorithm execution and do not assume any barriers in between accumulating labels (hence there is no superstep involved).

2.3. Graph Coloring

A *vertex-coloring* of a graph G finds an assignment of color to each vertex v in vertex set V such that no two adjacent vertices have the same color. The *graph-coloring* (GC) problem asks to find a vertex-coloring which uses as few colors as possible. The problem of finding an optimal coloring, $\chi(G)$, of a graph is NP-complete [53]. However, several

Alg. 2: SSSP Algorithm in KLA paradigm

In : Graph $\mathcal{G} = \langle V, E \rangle$, source s, Parameter k

$\forall\, v \in V$: $owner[v]$ = rank that owns v

Out: $\forall\, v \in V$: $distance[v]$ = distance of v

```
1  distance[v] ← ∞, ∀v ∈ V, i ← 0;
2  B ← Set of two buckets, for vertices within the current level range and next level
     range based on k;
3  enqueue (s, 0) ← B_i;
4  k_current ← 1;
5  message handler explore(Vertex w, distance d, level l)
6      newindex ← l mod 2;
7      currentindex ← (k * i) mod 2;
8      if d < distance[w] & currentindex != newindex then
9          enqueue (w, d, l) ← B_newindex;
10     else
11         enqueue (w, d, l) ← B_currentindex;
12 while B not empty do
13     while B_i not empty do
14         active message epoch
15             parallel foreach v ∈ B_i do
16                 if Update distance[v] then
17                     parallel foreach neighbor w of v do
18                         d = distance[v] + weight(v, w);
19                         k_current += 1;
20                         send explore(w, d, k_current) to owner(w);
21     i ← i + 1 mod 2;
```

heuristics-based approaches work well in practice. A vertex v is called a *Grundy vertex* if v is colored with the smallest color not taken by any neighbor. A *Grundy coloring* of G is one in which every vertex is a Grundy vertex [34]. A vertex v is called *properly colored* if for all $i \in neighbor(v), color(i) \neq color(v)$. Grundy coloring is a proper coloring of a graph. The minimum number of colors (*color classes* or *independent subsets*) needed to properly color a graph G is called the *chromatic number* of G, $\chi(G)$ and is a NP-hard problem. Grundy coloring always results in k colors where $\chi(G) \leq k \leq (\Gamma + 1)$ for a graph G with maximum degree of Γ. In this thesis, we are interested in Grundy coloring.

2.3.1. Ordering Heuristics. Since finding an optimal coloring is NP-hard, several greedy algorithms have been designed. Such algorithms employ different vertex ordering heuristics [62] to decide which vertex to color first in different coloring algorithms. This ordering is fixed before the algorithm starts execution. We call this **pre-execution ordering**. For example, the first-fit heuristic [75] colors vertices in the order they appear in the input graph representation. The random ordering heuristic [66] colors vertices in a uniformly random order. The largest-degree-first ordering heuristic [107] colors vertices in the order of decreasing degree. The incidence-degree ordering heuristic [35] iteratively colors an uncolored vertex with the largest number of colored neighbors. The smallest-degree-last ordering heuristic [11, 83] colors the vertices in the order induced by first removing all the lowest-degree vertices from the graph, then recursively coloring the resulting graph, and finally coloring the removed vertices. The saturation-degree ordering heuristic [20] iteratively colors an uncolored vertex whose colored neighbors use the largest number of distinct colors.

For both Jones-Plassmann and our algorithms, we consider first-fit and random ordering heuristics to assign order to vertices for coloring. In distributed setting, these heuristics have the lowest overhead and work well in practice with reasonable color quality. The resultant ordering is used to decide which vertex to color first. Imposing an ordering, in essence, creates a predecessor-successor relationship between vertices. In the following discussion, we refer to vertices with no predecessor as *roots*.

Alg. 3: Jones-Plassmann coloring algorithm

In : Graph $\mathcal{G} = \langle V, E \rangle$,

$\forall\, v \in V$: $owner[v]$ = rank that owns v

```
1  procedure Main()
2      foreach v ∈ V do
3          if predecessorCount[v] > 0 then
4              Allocate memory for predecessorColors[v] based on predecessorCount[v];
5      active message epoch
6          parallel foreach v ∈ V do
7              if owner[v] = this rank then
8                  Visit-root(v);
9  procedure Visit-root(Vertex r)
10     if predecessorCount[r] = 0 then
11         color[r] ← 0 ;
12         parallel foreach neighbor v of r do
13             if id[v] > id[r] then
14                 send Visit(v, color[r]) to owner(v);
15 message handler Visit(Vertex v, Color c)
16     predecessorColored[v]++;
17     predecessorColors[v][predecessorColored[v]] ← c ;
18     if predecessorColored[v] = predecessorCount[v] then
19         color[v] ← findMinAvailableColor(v) ;
20         parallel foreach neighbor w of v do
21             if id[w] > id[v] then
22                 send Visit(w, color[v]) to owner(w);
```

2.3.2. Jones-Plassmann Coloring algorithm. We have chosen Jones-Plassmann Coloring algorithm as our baseline algorithm because only recently it has been proven to be efficient in distributed setting [101], both in terms of execution time and optimality of the result [35] . This algorithm is based on asynchronous push mechanism, where vertices push their states to the successors.

Jones-Plassmann algorithm (Alg. 3) works as follows: each vertex maintains a list of colors taken by the predecessors to keep track of how many predecessors have been colored so far (Ln. 4). The algorithm starts by assigning color 0 to the roots (Ln. 11) and sending out the information to all their successors (Ln. 14). When a vertex v finishes obtaining all predecessors' colors (Ln. 18), it starts searching for a minimal available color. When it finds an available minimal color value, it assigns the color to itself (Ln. 19). If all the colors in the range $0, 1, 2, \cdots, predecessorCount[v] - 1$ are taken, the vertex assign $predecessorCount[v]$ as its color. Once colored, the vertex sends its color information to all its successors (Ln. 22).

2.4. Active Pebbles Support for Asynchronous Graph Execution

Our main implementation of the baseline algorithms and our algorithms is based on the Active Pebbles model [110]. Rather that bringing in (pulling) data for computation, AP model sends the computation to the target where data is placed. The communication is based on fine-grained active messages (AM) that are called *pebbles*. In an active message [105] framework such as in AP, messages are sent explicitly but receives are implicit. Here, since a pre-registered receive handler knows a priori the address of user-level handler, the computation can be integrated with less software overhead. Active pebbles are unordered except by the termination detection. The granularity of graph operations in AP model can be as small as an operation on a vertex or an operation on an edge and thus can be expressed in their natural form.

AP model consists of two components: a programming model and an execution model. The *programming model* consists of globally-addressable light-weight objects, pebbles and

targets. While fine-grained pebbles make it easier to intuitively express graph operations (such as vertex-state update) in AP model, runtime intervention is needed to ensure several optimizations. These optimizations constitute the *execution model* of AP and are implemented in the AM++ [108] runtime. These optimizations include message reduction (sender-side aggregation), message coalescing (receiver-side scatter) and active routing.

Since, in active pebbles model, computation is carried out on local data, challenges such as maintaining global memory consistency and designing sophisticated vertex locking protocol for concurrent updates are eliminated. Guaranteeing such memory consistency is required, for example in Powergraph [55], for both synchronous and asynchronous execution. AP model, on the other hand, only depends on local atomic operations available on native architecture to update a vertex property value. This is specially helpful for designing high-performance asynchronous unordered graph algorithms.

In addition to retaining AM features, AP model has a distinguishing feature: message handlers in AP can directly generate and send arbitrary messages to any destination, to unbounded depth. Also message handlers are not restricted only to send replies. This feature differentiates AP from other low-level active message system such as GASnet. This has an added benefit of eliminating the need for application-level message buffering. Irregular application such as graph applications can have variable-length depth of computation during the execution of an algorithm. Such unbounded (variable-length) computation is well-represented by such extended active-message handler. In particular, asynchronous, unordered graph algorithms can trigger chained computation and AP message-handlers lend themselves naturally to such mode of computation by allowing arbitrary operations within handlers.

However, such added flexibility can complicate termination detection. To handle such arbitrary chains of nested messages, unlimited-depth termination detection algorithm is needed. Currently a four-counter based algorithm similar to the one proposed by Sinha, Kal'e and Ramkumar [103] (SKR) is implemented in the AM++ runtime.

An active pebble message spawns tasks asynchronously by executing a message handler on the rank where data resides. Currently the underlying transport of AM++ is based on MPI and the asynchronous send operations are implemented with `MPI_Isend`s and `MPI_Irecv`s paired with `MPI_Testsome`.

2.5. Ordering in Graph Algorithms

Asynchronous execution of graph algorithms results in unordered label-correcting algorithms. However, such unrestricted execution of workitems can result in work explosion and redundant work. To circumvent this problem, ordering can be imposed on tasks to decide which task to process first. To separate the notion of ordering from actual processing of a vertex state, recently Abstract Graph Machine (AGM) [46] has been proposed. An AGM represents a graph algorithm as two distinct components: a *processing* function responsible for the actual computation on a vertex for a particular algorithm and an *ordering* of the tasks that captures the order in which these computations are executed. For example, in a Single-Source Shortest Path (SSSP) algorithm, tasks can be ordered based on the distance from the source. The work ordering relation is a strict weak ordering. This relation partitions tasks into ordered equivalence classes, where tasks (work) within an equivalence class can be executed in parallel in any order, however tasks within disjoint equivalence classes are executed according to an ordering imposed by the strict weak ordering relation.

Ordering can be enforced at different spatial levels of architecture such as globally on the whole distributed setting (Dijkstra's priority queue for SSSP for example) , node-level (process), Non-Uniform Memory Access (NUMA) domain-level, and thread-level. While placing global ordering can be beneficial in terms of work optimality, this can seriously restrict the available parallelism. When combined with asynchrony, one would strive to find a suitable spatial level or composition of several levels for ordering. The reasonable level of ordering helps to find a good balance between the amount of work executed and available parallelism. With the right priority measurement, such ordering can be

adequate enough to approximate the global ordering but would not be too restrictive to limit parallelism. For example, a framework has been proposed in [46] that captures the essence of the spatial ordering aspect of an algorithm while leaving temporal scheduling aspect to the runtime such as active pebbles execution model.

2.6. Conclusion

In this chapter, we have discussed the baseline algorithms for three different graph kernels: Single-source shortest paths, connected components and graph coloring. We compare the performance of our algorithms with these baseline algorithms in the following chapters. Moreover, we have given an overview of Active pebbles model and have discussed how Active pebbles enables asynchronous graph execution with various runtime supports such as termination detection and unbounded-depth active message handlers.

CHAPTER 3

Synchronization-Avoiding Graph Algorithms

One of the main goals of this thesis is to design synchronization-avoiding graph algorithms. As discussed in Chapter 1, many traditional approaches in designing graph algorithms suffer from straggler effect due to inherent global and vertex-centric barriers embedded in the algorithms. Such barriers can restrain an algorithm from unveiling the full potential of an underlying distributed runtime that supports fine-grained threading and asynchronous communication.

In an effort to avoid global synchronization and vertex-centric barriers, we combine Active Pebbles [110] asynchronous graph execution model with low-overhead thread-level ordering and design highly scalable unordered graph algorithms. For temporal scheduling we rely on active message based AM++ runtime [108], whereas for spatial scheduling we rely on thread-level priority queues. In this chapter, we discuss our algorithms at length. We refer to these algorithms as *Distributed Control* (*DC*) algorithms.

3.1. Classification of Algorithms

Let us assume that an algorithm is evaluating a property P for vertices with each vertex v's property value denoted by p_v. To design synchronization-avoiding, locally-ordered, label-correcting algorithms, we broadly categorize them into two main classes: algorithms performing monotonic updates to p_v and algorithms performing non-monotonic updates to p_v. In the monotonic update case, p_v either monotonically increases or decreases. On the other hand, non-monotonic property updates do not strictly increase or decrease p_v. An example of monotonic update is the SSSP problem where distance from the source to a vertex decreases as the algorithm progresses (or remains at ∞ if the vertex is disconnected from the source). Breadth-first search (BFS) can be considered as a special case of SSSP

where weight of each edge is set to zero. Other examples of applications with monotonic updates include connected component, data-driven pagerank, single-source widest path, minimum-spanning tree etc. Graph (vertex) coloring can be cited as an example of non-monotonic update. Here a minimum color is chosen within a range of available colors during a particular iteration of an algorithm. However, in subsequent iterations, the color value can either go up or down depending on the predecessor colors. Other examples of non-monotonic update is the maximal independent set (MIS) calculation, edge coloring etc. Graph problems in each of these categories can be solved with our approach discussed in Sec. 3.3.

3.2. Execution-time Ordering By Priority

Elimination of global synchronization and vertex-centric barriers results in unordered algorithms. Unordered algorithms do not require tasks to be executed in any particular order for the correctness of the final result. Since there are no dependencies among tasks, this type of label-correcting algorithms are particularly suited for optimistic (speculative) parallelization and asynchronous execution. However, such unconstraint execution of tasks in unordered algorithms may result in work explosion and sub-optimal tasks. To reduce the amount of sub-optimal task execution in unordered algorithms, in our algorithms, updates are ordered according to a *priority* measurement. The scheduler uses this priority metric to decide which task to execute during algorithm execution (*execution-time ordering*).

To formulate such priority measures, we consider algorithms with monotonic and non-monotonic updates separately. In the algorithms with monotonic updates, potential updates can be ordered by giving priority to updates that contain better values (either smaller or larger values depending on the problem) compared to others. For example, distance from the source can be considered as a priority measurement for the SSSP algorithms. For the connected component algorithms, priority can be given to the tasks with smaller component numbers.

However, with a non-monotonic update function, it may not be obvious what metrics can be considered as the priority measures for ordering tasks. In this case, one may have to impose ordering by combining two or more metrics. We differentiate *execution-time* ordering from *pre-execution* ordering. *Pre-execution ordering* is fixed before an algorithm starts. This ordering is imposed by identifiers (higher vs. lower ID compared to the neighbors), degrees (higher degree vs. lower degree compared to the neighbors) etc. of vertices and forms predecessor-successor relationships among them. Pre-execution ordering helps coloring algorithms to decide which set of vertices will be considered as roots (i.e.have no predecessor) to start an algorithm. One metric for *execution-time ordering* of messages is the distance of the current vertex from such roots in a DAG. We call such DAG structure *execution DAG* that unfolds during the execution of an algorithm, when messages are passed from predecessors to successors. The distance priority metric can be combined with another metric specific to a particular problem in hand (for example a smaller color value in the graph coloring problem) to form a composite priority metric for ordering tasks in the non-monotonic case.

3.3. Algorithms

Our algorithms for monotonic and non-monotonic updates are shown in Alg. 4 and Alg. 7 respectively.

3.3.1. Initialization step of the algorithms. In both algorithms, each rank maintains a set of thread-local priority queues (Ln. 1 in Alg. 4 or Ln. 1 in Alg. 7). These priority queues approximates the global ordering without incurring high overhead. For each vertex, the property-map data-structure, $property[v]$, maintains the vertex property value (distance, component no., color etc.) to be determined. At the beginning of the algorithm, $property[v], \forall v \in V$ is set to an initial value (Ln. 2 in Alg. 4 or Ln. 2 in Alg. 7). For example, in SSSP algorithm, distance to all the vertices from the source is set to ∞. For connected component algorithm, each vertex belongs to its own component. For graph coloring algorithm, each vertex color is set to (predecessor count + 1).

Alg. 4: Parallel Active Pebble algorithm with monotonic update values calculated at the sender side.

In : Graph $\mathcal{G} = \langle V, E \rangle$, set of roots, R

$\forall\, v \in V$: $owner[v]$ = rank that owns v,

activeMessageType $m = (v, p, \rho)$ where p =

tentative property value p and ρ = priority

Out: $\forall\, v \in V$: $property[v]$ = property of v

1 $Q \leftarrow$ set of empty per-thread priority queues;

2 $property[v] \leftarrow initial\ value, \forall v \in V$;

3 $activeCount = finishCount = 0$;

4 $property[r] \leftarrow 0, \forall r \in R$;

5 $\rho \leftarrow calculateNewPriority(r, r, property[r]), \forall r \in R$;

6 enqueue $(r, property[r], \rho) \rightarrow Q, \forall r \in R$;

7 $activeCount += |r|$;

8 startActiveMessageEpoch(); startTerminationDetection();

9 **active message epoch**

10 **parallel foreach** $q \in Q$ **do**

11 **if** q not empty **then**

12 $(v, p) \leftarrow$ dequeue q ;

13 $finishCount{+}{+}$;

14 tryMonotonicUpdate(v, p);

Alg. 5: Monotonic update function

```
1  procedure tryMonotonicUpdate(Vertex v, Property p)
2      while p < property[v] do
3          atomic
4              oldProperty ← load(property[v]);
5          atomic
6              success ← CAS(property[v], oldProperty, p) ;
7          if success then
8              parallel foreach neighbor w of v do
9                  newProperty ← calcNewProp(v, w, p) ;
10                 ρ ← calculateNewPriority(v, w, p);
11                 m ← buildMessage(w, newProperty, ρ);
12                 activeCount++;
13                 send explore(m) to owner(w);
14             break ;
15 message handler explore(activeMessageType m)
16     enqueue m → q, q ∈ Q;
```

In addition, if there are specific roots, $r \in R \subset V$ to start from, property values of these roots are set to zero (Ln. 4 in Alg. 4 or Ln. 4 in Alg. 7). For example, in SSSP algorithm, the source sets its distance to itself to zero. In coloring algorithm, all the vertices with no predecessor set their colors to 0. These updates are then enqueued in the thread-local priority queues (Ln. 6 in Alg. 4 or Ln. 6 in Alg. 7) before propagating to the neighbors.

In the connected component algorithm, all the vertices are considered root and thus are directly pushed into the queues.

3.3.2. Algorithms With Monotonic Updates. Our algorithm with monotonic updates (Alg. 4) starts with an active message epoch on each rank (Ln. 9 in Alg. 4). Within the

epoch, each thread checks its priority queue to see whether there is any task (message) to process. If a task is successfully dequeued from the thread-local queue (Ln. 12 in Alg. 4), the current thread tries to update the current property value, $property[v]$, of vertex v with the value contained in the task (Ln. 14 in Alg. 4 and Alg. 5). This attempt for update is made with an compare-and-swap (CAS) atomic operation (Ln. 6 in Alg. 5). If this atomic operation succeeds, a new property value for the neighbors are computed (Ln. 9 in Alg. 5) (for example, Alg. 6 for SSSP) and this update is sent asynchronously to all the owners of the neighbors (Ln. 13 in Alg. 5).

On receipt of the messages, message handlers insert the messages into thread-local priority queues of the recipient threads (Ln. 16 in Alg. 5). The algorithm terminates when there is no messages left in the thread-local priority queues. It is to be noted that, in monotonic update case, the potential update value of a neighbor's property is calculated on the sender side. This update calculation is different from algorithms with non-monotonic updates described later. Additionally, the message handler shown in Alg. 5 is an user-level message handler. A runtime Active Pebbles (AP) message handler is responsible for spawning a user-level handler for each message in the received coalesced buffer. Until flow control kicks in, this runtime handler can keep spawning user-level handlers, thus can trigger nested computations (and subsequent sends). This functionality of AP message handler is different from traditional active message handlers, the later being only capable of sending acknowledgments.

Alg. 6: Procedure to calculate distance.

```
1 procedure calculateNewDistance(Vertex s, Vertex t, Distance p)
2     return weight(s, t) + p ;
```

3.3.3. Algorithms With Non-monotonic Updates. Algorithms with non-monotonic updates (Alg. 7) have two distinctive features: a local termination counter for each vertex (pending task counter in Fig. 3.1 or $localTermCounter$ in Alg. 7) and a composite priority metric to impose ordering on tasks. Local termination counters are required to eliminate

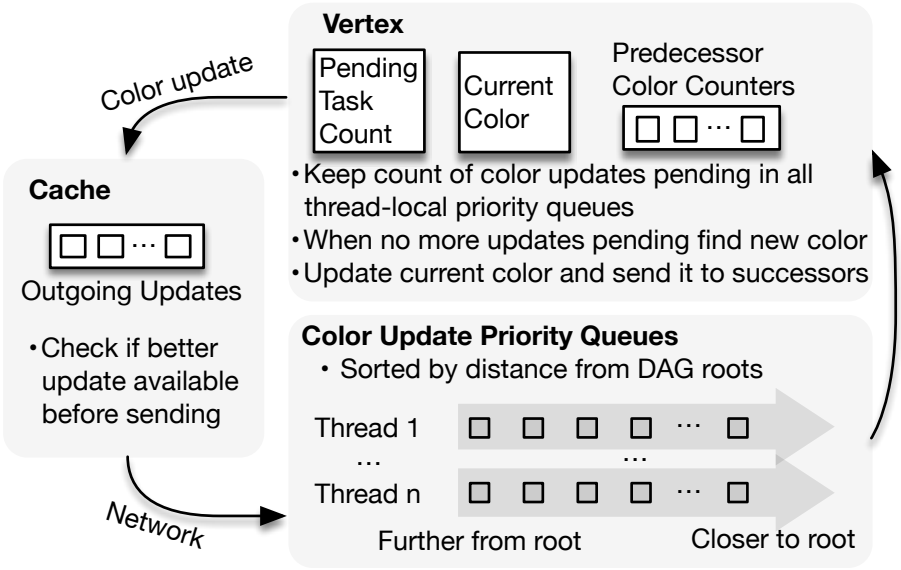


FIGURE 3.1. Overview of Distributed Control (DC) coloring algorithm.

vertex-centric barriers. Instead of waiting for all the predecessors to obtain their final property value, a vertex can optimistically go ahead and try to update its property whenever its local termination counter reaches a value of zero. This counter is incremented when a message is received from a predecessor and is enqueued into a thread-local priority queue (Ln. 18 in Alg. 8). The counter gets decremented when the message is taken out of the priority queue (Ln. 15 in Alg. 7). With this technique, vertex-centric barriers (implicitly dictated by the dependence on the final property values of the predecessors) are eliminated.

Without loss of generality, we will discuss the algorithm in terms of a concrete graph application: graph coloring (Fig. 3.1). The algorithm also starts with an active message epoch on each rank (Ln. 10 in Alg. 7). Within the epoch, each thread tries to dequeue a message received from a predecessor from the thread-local priority queue (Ln. 13 in Alg. 7). A message received from a predecessor contains two pieces of information: predecessor's old color and predecessor's new color. After dequeuing the message, $tryNonMonotonicUpdate$ procedure calls in $calcNewProp$ procedure (Ln. 2 in Alg. 8) (Since the property value to be determined is color, we call the equivalent procedure $calculateNewColor$ (Alg. 9)). On Lns. 2–3 of Alg. 9, the receiving vertex v decrements the color counter of the predecessor's

Alg. 7: Parallel Active Pebble algorithm for non-monotonic update values calculated at the receiver side.

In : Graph $\mathcal{G} = \langle V, E \rangle$, set of roots, R

$\forall\, v \in V$: $owner[v]$ = rank that owns v,

activeMessageType $m = (v, o, n, \rho)$ where o and n are old

and new property value and ρ = priority,

$\forall\, v \in V$: $localTermCounter[v]$ = local

termination counter for v at $owner[v]$

Out: $\forall\, v \in V$: $property[v]$ = property of v

$Q \leftarrow$ set of empty per-thread priority queues;

$property[v] \leftarrow initial\ value,\ \forall v \in V$;

$activeCount = finishCount = 0$;

$property[r] \leftarrow 0,\ \forall r \in R$;

$\rho \leftarrow calculateNewPriority(r),\ \forall r \in R$;

enqueue $(r, property[r], \rho) \rightarrow Q,\ \forall r \in R$;

$activeCount\ +\!= |r|$;

$localTermCounter[r]{+}{+}\ \forall r \in R$;

startActiveMessageEpoch(); startTerminationDetection();

active message epoch

  parallel foreach $q \in Q$ **do**

    if q not empty **then**

      $(v, o, n) \leftarrow$ dequeue q ;

      $finishCount{+}{+}$;

      $localTermCounter[v]{-}{-}$;

      tryNonMonotonicUpdate(v, o, n);

Alg. 8: Non-monotonic update function.

```
1  procedure tryNonMonotonicUpdate(Vertex v, Property o, Property n)
2      newProperty ← calcNewProp(v, o, n) ;
3      while newProperty < property[v] do
4          atomic
5              oldProperty ← load(property[v]);
6          atomic
7              success ←
8                  CAS(property[v], oldProperty, newProperty)
9          if success then
10             ρ ← calculateNewPriority(v);
11             parallel foreach neighbor w of v do
12                 m ← buildMessage(w, newProperty, ρ);
13                 activeCount + +;
14                 send explore(m) to owner(w);
15             break ;
16 message handler explore(activeMessageType m)
17     enqueue m → q, q ∈ Q;
18     localTermCounter[v]++;
```

old color o, $predcolor[v][o]$, by one and increments the new color n count, $predcolor[v][n]$ by one. Next it checks whether the new color n of the predecessor is the same as its own color or whether the local termination counter for the current vertex v has reached a value of zero (i.e. no message is waiting to be processed in the thread-local priority queues targeted for v) (Ln. 5 in Alg. 9). In either case, a search is triggered for a new available color for vertex v (Ln. 6 in Alg. 9). Once a suitable minimum color value is found, the worker thread tries to update the vertex color with a CAS atomic operation (Ln. 6 in Alg. 8). If successful, a new priority is calculated. The priority is calculated based on the current vertex's distance from the roots (i.e. from the vertices with no predecessor) and its color value (Ln. 2 in Alg. 8). The smaller the distance and color value, the higher the priority of the message.

The observation behind choosing distance from the root as a priority metric is that successors of vertices closer to the roots can not proceed to obtain the right color value until the predecessors propagate their color information. This distance metric is combined with smaller color value because vertices are colored with minimum available color and propagating the information about which smaller color values have already been taken by the predecessors helps the search for new available color to choose the right minimum color earlier. Note that, in contrast to the monotonic update case, property update values are calculated on the receiver side for non-monotonic updates.

Once the new color update and priority is calculated, the information is sent asynchronously to all the neighbors along with the deserted old color value (Ln. 14 in Alg. 8). The receiving message handler enqueues the message into one of the thread-local priority queues for processing (Ln. 17 in Alg. 8).

3.3.4. Priority Based On Distance From Roots In The Execution DAG. Graph coloring is one of the examples where one of the priority metric is the distance of a vertex from the root in the execution DAG. Another example where this priority metric can be used is the Maximal Independent Set (MIS) algorithm discussed in [68].Here vertices are classified into two categories: vertices in the MIS (labeled FIX1) and vertices not in the

Alg. 9: Procedure to calculate new color.

```
In  : predcolor[v] = colors of predecessors of v
Out : newcolor, minimum available color
1 procedure calculateNewColor(Vertex v, Color o, Color n)
2     predcolor[v][o]−− ;
3     predcolor[v][n]++ ;
4     newcolor ← ∞ ;
5     if (localTermCounter[v] == 0) || (n == color[v]) then
6         newcolor ← min{i | predcolor[v][i] == 0} ;
7     else
8         newcolor ← total_predecessor_count[v] + 1;
```

Alg. 10: Procedure to calculate new priority for coloring.

```
1 procedure calculateNewPriority(Vertex v)
2     makeCompositePriority(distanceFromRoot(v), color[v]);
```

MIS (labeled FIX0). If a vertex is labeled as FIX1, all its neighbors are labeled as FIX0. Messages containing FIX1 are processed immediately. Vertices which are closer to roots and currently labeled as FIX0 are given priority over the vertices that are further from the root. This is based on the same observation that, successors can not proceed with the right labeling if they have to wait on the predecessors to obtain the right label. MIS is a simplified version of graph coloring problem, which operates only with two colors.

3.3.5. Message Caching for Coloring Algorithm. In order to reduce the propagation of sub-optimal work, we cache messages destined for the successors before sending them (Fig. 3.1). For this purpose, we have implemented a customized reduction cache. Before processing any element from the thread-local priority queues containing predecessor color information, the algorithm attempts to send messages to the successors that have been cached. When the color information of a vertex is popped from the cache , a check is

performed to see whether the vertex is already updated with a better color or whether the color has not been changed since last update. If both of the conditions fail, the current vertex color is recorded as the last color sent and the updated color information is propagated to its successors.

3.4. Algorithmic Requirements

In general, to ensure correctness of asynchronous algorithms, the update operation on a vertex property must be correct in the presence out-of-order message execution. For monotonic updates, this requirement is fulfilled by the monotonicity of updates, and for non-monotonic algorithms it results from following an execution DAG. Here we briefly discuss correctness and termination of our algorithms. We assume no message has been lost during the execution of an algorithm.

3.4.1. Correctness.

3.4.1.1. *Monotonic case.* For monotonic updates, correctness is ensured by the monotonicity property. For monotonic updates, with each iteration of the update operation, the absolute difference between the final property value and the current property value p_v decreases. Since messages are propagated to the neighbors whenever an update occurs, it is possible that the delivery of a better value can get delayed. However, as we assume that no faults occur during message propagation, all messages are delivered eventually, and each vertex v gets chance to update p_v based on the received messages. The lossless message propagation along with the invariant that updates to p_v only happen when the received value is monotonically decreasing (increasing) and that the value never changes once it reaches final value ensure correctness.

3.4.1.2. *Non-monotonic case.* We show that our graph coloring algorithm achieves correct Grundy coloring by contradiction. Let us assume that the algorithm does not produce correct result. Then there exists at least one vertex that either ended up with the wrong minimum color value within permissible range $[0, \Gamma + 1]$ or the current color is in direct conflict with the color value of its predecessor(s). However, the following facts prevent this

situation. Color counters are incremented and decremented for $newColor$ and $oldColor$ of the predecessors to keep track of which color group has been joined or left by the predecessors (Lns. 2–3 in Alg. 9). When the $localTermCounter$ reaches a value of zero or when the current color of a vertex is taken by a predecessor (Ln. 5 in Alg. 9), the previous counters allow a vertex to find the right minimum available color. All the changes from the predecessors are tracked by these counters. Speculative execution can only result in temporary updates to the successor colors based on wrong predecessor colors. But as soon as a predecessor gets a better color, the successors are notified with the updates to trigger corrective measures. Consequently, the vertices in each level of the execution DAG will ultimately stabilize with the correct Grundy color.

3.4.2. Termination. Our termination detection works by maintaining two monotonically increasing counters $activeCount$ and $finishCount$ (Ln. 3 in Alg. 4 or Ln. 3 in Alg. 7) on each rank. The initiator of messages increases the local $activeCount$ counter by the total number of recipients (neighbors) (Ln. 12 in Alg. 5 or Ln. 13 in Alg. 8). The $finishCount$ counter is incremented at the recipient when a message is dequeued from a thread-local priority queue (Ln. 13 in Alg. 4 or Ln. 14 in Alg. 7). The global quiescence is checked periodically by the AM++ runtime performing global reductions on these counters in two phases and checking whether their difference has reached a value of zero in subsequent phases (similar to the four-counter based algorithm proposed by Sinha et al. [103]). Once the algorithms converges, no more messages are generated and the difference between these two counters reaches a value of zero, resulting in the termination of the program.

3.5. Implementation on AM++ Runtime

For performance, our algorithmic approach for avoiding synchrony must satisfy two conflicting needs. On the one hand, we want the maximum ordering we can achieve, so we rely on the underlying runtime system to deliver tasks to the appropriate private workset as soon as possible. On the other hand, quick delivery comes at a cost: the accumulative costs of network sends overhead, the necessity for frequent polling, frequent context switches

when handling small tasks, and so on, add up to a significant overhead. To balance these needs, we use the AM++ [108] runtime, which supports fine-grained parallelism of active messages with communication optimization techniques such as scalable addressing, active routing, message coalescing, message reduction, and termination detection.

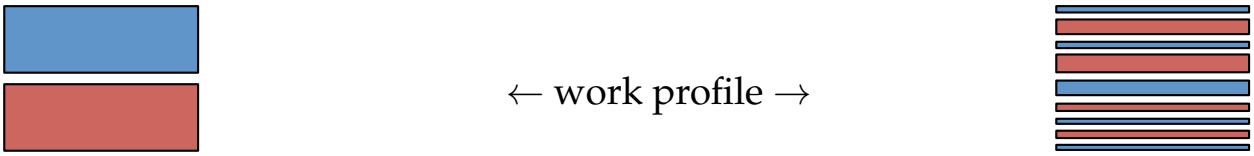
AM++ is based on the Active Pebbles (AP) model [109]. As discussed earlier, at the core of the AP model are *pebbles*, lightweight active messages that are sent explicitly but received implicitly. The implicit receive mechanism is based on *handlers*, which are user-defined functions that are executed in response to the received pebbles.

AM++ provides an interface that can be executed by many workers (single-program, multiple data—SPMD). Each worker can execute independently, and when it calls AM++ interfaces it may execute tasks from the AM++ *task queue*, which schedules tasks such as network polling, buffer flushing, and pending handlers. At the lowest level, pebbles are sent and received using *transports* that encapsulate all low level AM++ functionality such as network communication and termination detection. Currently, the low-level network transport of AM++ is built atop of MPI, but none of the MPI interfaces are exposed to the programmer, and AM++ has supported other transports in the past.

In order to send and handle active pebbles, individual *message types* must be registered with the transport. A transport, given the type of data being sent and the type of the message handler, can create a complete message type object. To decrease the overhead of sending many small messages, AM++ performs *message coalescing*, combining multiple pebbles sent to the same destination into a single, larger message. In the current implementation, a buffer of messages is kept by each node for each message type that uses coalescing, for each possible destination. The size of coalescing buffers is determined by the maximum number of pebbles to be coalesced and the pebble size. Messages are appended to the buffer, and the entire buffer is sent when it becomes full or it is *flushed* when there is no more activity. Message coalescing increases the rate and decreases the overhead at which small messages can be sent over a network. The transport layer costs (bookkeeping, message injection) are amortized over many messages at some cost to latency. However,

```
1 {
2   epoch e;
3   // work
4 }
```

```
1 epoch e;
2 while(!e.end()) {
3   // work
4 }
```



(A) Scoped epoch with two distinct phases of work.

(B) End-epoch test with interleaving of work and progress.

FIGURE 3.2. Two epoch execution models and the interleaving of work (blue) with AM++ progress (red).

this cost is expected to be offset by large problem scales that depend on throughput more than on latency.

3.5.1. Epochs. The AM++ runtime is based on *epochs*, in which messages can be sent and during which all the resulting handlers are executed (*termination detection*). All workers must enter and exit the epoch collectively, with the exit possible only after all the handlers in the epoch are executed. AM++ guarantees that the handlers for all the messages sent within a given epoch will have completed by the end of that epoch. In addition, it also guarantees that calling end of an epoch test interface will progress AM++ execution. Because AM++ allows handlers to send arbitrary new messages, it relies on a termination detection algorithm to discover when no more handlers are left to execute and no more pebbles are in flight.

In general, AM++ workers perform two kinds of work: the worker's "private" work, and AM++ progress that can occur any time an AM++ interface is called. A thread's private work includes tasks such as local bookkeeping or preparing for an epoch. AM++ progress consists of handler execution, crucial to algorithm progress, and bookkeeping and maintenance tasks such as network polling, buffer flushing, and termination detection. In general, an AM++ program consists of general setup, including creating a transport

and registering message types with the transport along with required properties such as coalescing and object-based addressing. After all the necessary machinery is created, an AM++ program executes one or more epochs. Epochs can be executed in two significantly different ways. Figure 3.2a shows a *scoped epoch* in which application work is executed first, and when its scope ends, AM++ continuously executes progress, including member handlers, until no more pebbles are pending. The application can still send messages, and progress can be executed when messages are sent, but, in general, progress is only guaranteed to occur at the end of the scoped epoch resulting in a two-part work pattern (red in the figure is application work and blue is the progress at the end). Figure 3.2b shows the *end-epoch test* model, in which application executes some work in a loop, testing for the end of the epoch. This model allows an application to interleave its own work with AM++ progress, resulting in a pattern of potentially unequal periods of time spent in each portion of the work. Baseline algorithms are naturally based on the scoped epoch model. In a crucial difference, our algorithms rely on the end-epoch test model to execute whole algorithm without unnecessary gaps in work.

3.6. Experimental Results

3.6.1. Experimental Setup.

3.6.1.1. *Dataset.* We evaluate the performance of our algorithms with synthetic inputs as well as real world datasets.

Characteristics of synthetic graphs: To generate synthetic graphs, we employ the RMAT graph generator [73]. RMAT graph generator works by dividing the adjacency matrix of a graph into 4 separate quadrants. The probability of an edge between two vertices from different quadrants is specified by 4 parameters: a, b, c, and d. Parameters a and d specify community structures (sub-communities) in a graph. Connections among different sub-communities depend on parameters b and c. Also the skewness of degree distribution of vertices depends on a and d. Larger values of a indicates more skewed (in terms of presence of high-degree vertices) graph inputs.

We experiment with 4 types of synthetic graph inputs: Erdős-Rényi (RMAT-ER with $a = b = c = d = 0.25$), Graph500 [88] (with $a = 0.57, b = 0.19, c = 0.19, d = 0.05$), RMAT-G (with $a = 0.45, b = 0.15, c = 0.15, d = 0.25$), and RMAT-B (with $a = 0.55, b = 0.15, c = 0.15, d = 0.15$). As shown in [25], each of these inputs has different community structures and connections among such communities. Moreover, the vertex degree distribution pattern (skewness) varies across these graph inputs. For example, RMAT-ER has normal degree distribution with only one global maxima. All other RMAT graphs have several local maxima in the degree distribution plot, suggesting sub-communities embedded within these graphs. For coloring algorithm, we modified values of parameters b and c for RMAT-G so that it has denser connections between communities in the graph structures. We denote this graph as RMAT-$\widetilde{G}$ (with $a = 0.45, b = 0.25, c = 0.25, d = 0.05$). This enables us to experiment with graph structures requiring more colors due to denser connection among sub-communities. This is an important use case to demonstrate the effectiveness of optimistic execution such as ours over Jones-Plassmann algorithm. In our plots, a graph of scale x denotes a graph with 2^x vertices. Each vertex in the RMAT graphs has an average degree of 16 (directed), for a total of $2 * 16 * 2^x$ edges, considering undirected edges.

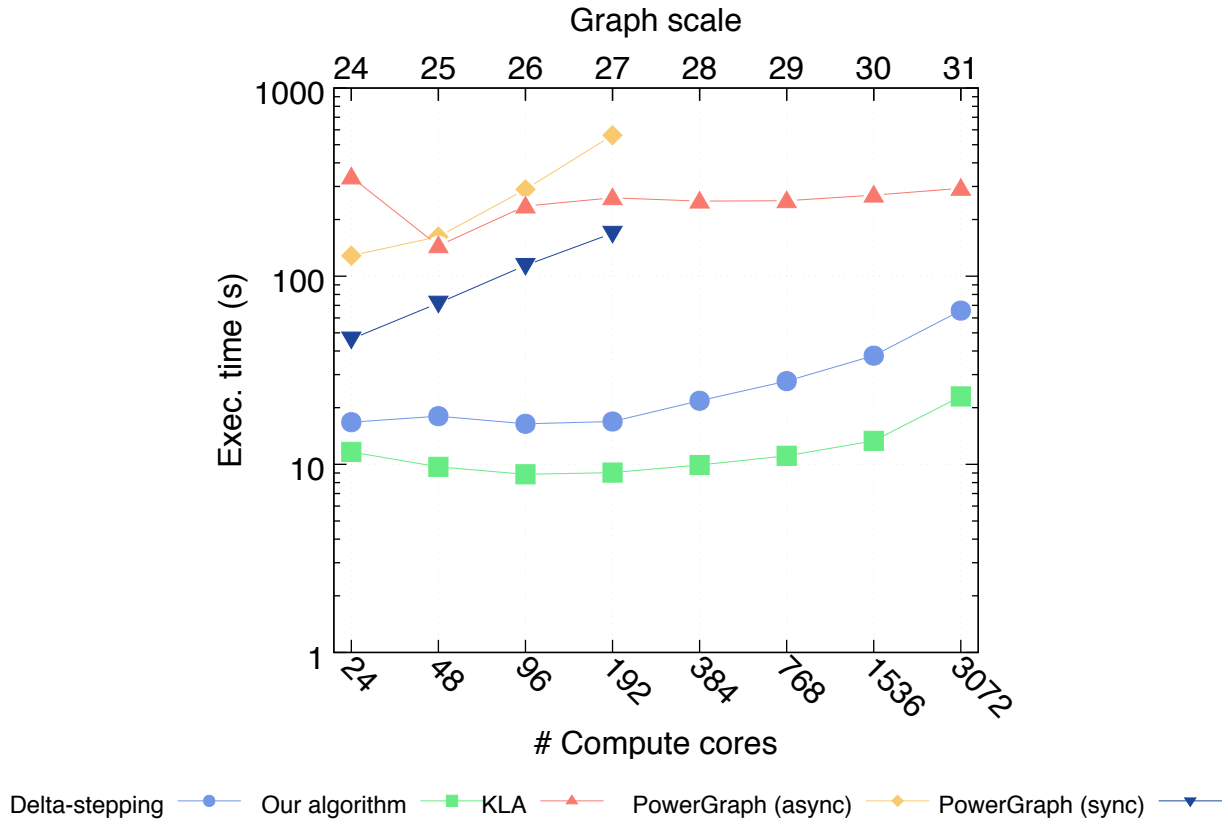


FIGURE 3.3. Weak scaling results for SSSP algorithms with RMAT-ER input.

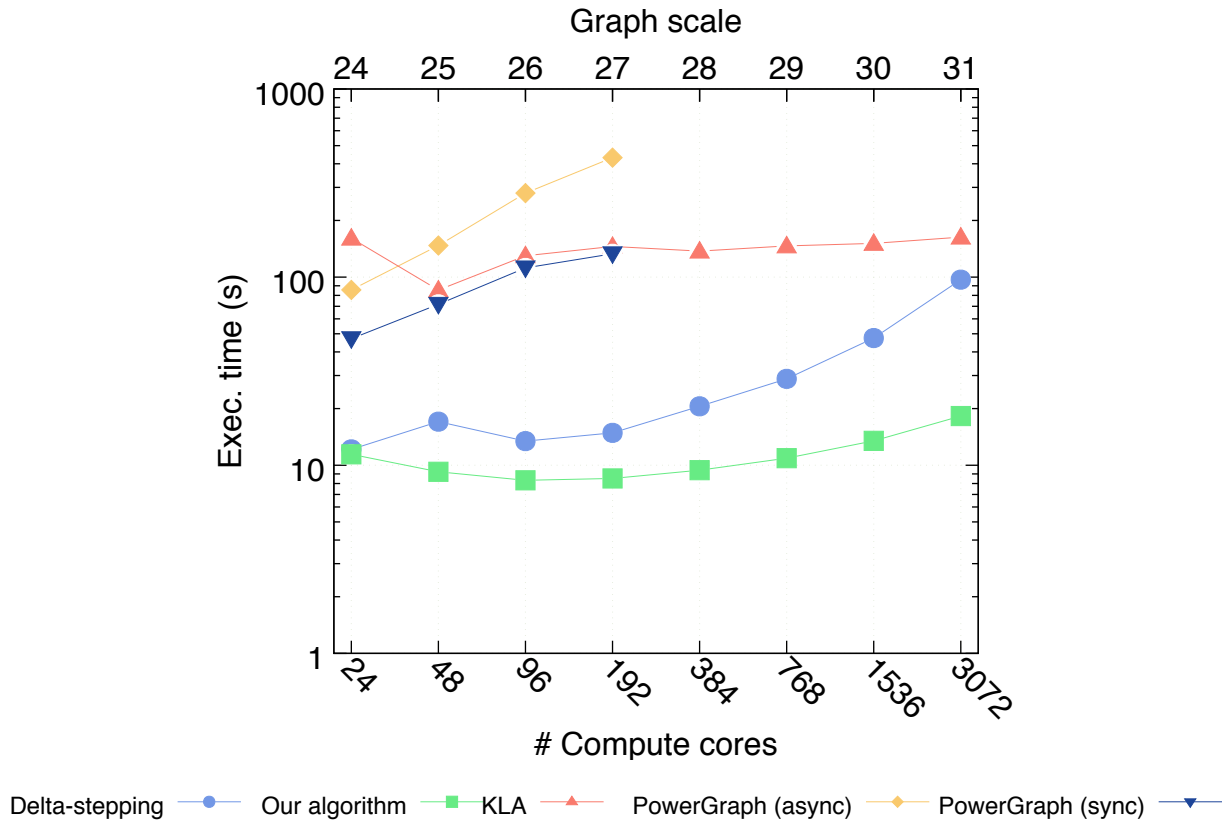


FIGURE 3.4. Weak scaling results for SSSP algorithms with RMAT-G input.

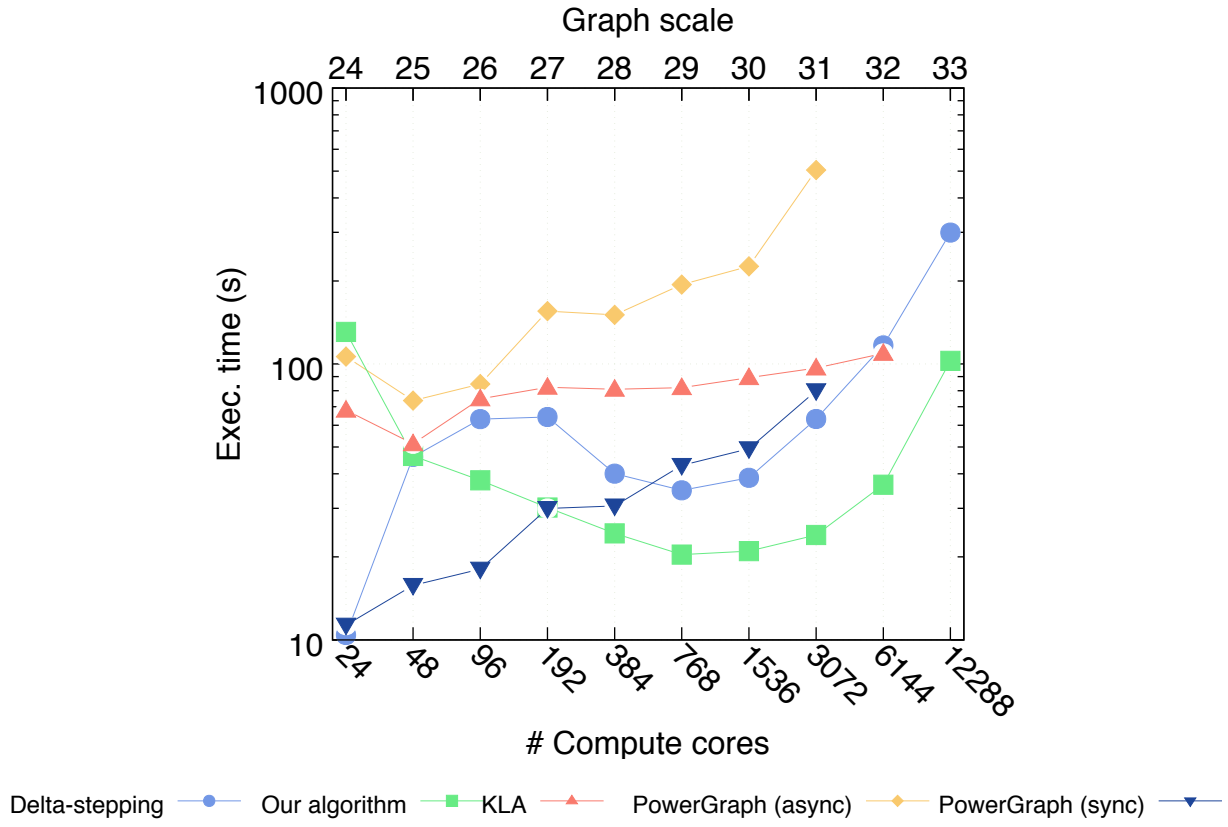


FIGURE 3.5. Weak scaling results for SSSP algorithms with Graph500 input.

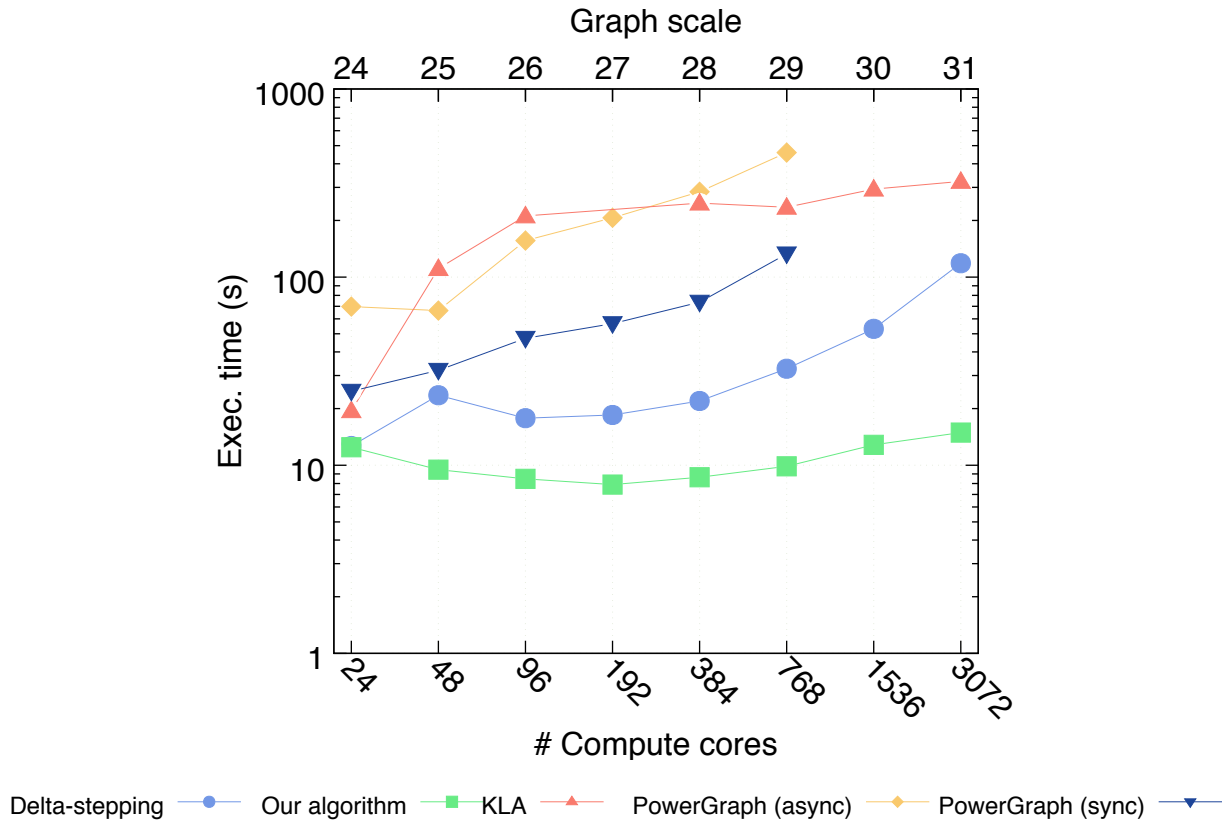


FIGURE 3.6. Weak scaling results for SSSP algorithms with RMAT-B input.

Real-world dataset. For our experiments, we have chosen a large set of real-world graph inputs, tabulated in Table 1 along with their characteristics. These inputs represent graphs with wide range of variations in diameters and clustering coefficients. Clustering co-efficient is a measure of community structure in graphs. We obtain these graphs from [8, 74, 38].

3.6.1.2. *Configuration.*

Hardware. We have conducted our experiments on a Cray XC30 system. Each compute node on the XC30 system consists of two Intel Xeon E5 12-core x86_64 2.3 GHz CPUs with hyper-threading enabled (up to 48 hardware threads per node) and of 64 GB of DDR3 RAM. All XC30 nodes are connected through the Cray Aries interconnect.

Compiler Options. We compiled our code with gcc 7.2.0 and with optimization level '-O3'. Additionally, single node experiments were run with networking turned on.

Comparison with Powergraph. We compare the performance of our implementations in AM++ with PowerGraph [55], a well-known distributed graph processing framework. This helps us to evaluate the efficiency (and coloring quality) of our algorithms. We performed our experiments with the publicly-available version 2.2 of PowerGraph [6]. PowerGraph processes vertex-centric programs in three phases: *Gather* (gather results from neighbors), *Apply* (compute new updates), and *Scatter* (propagate updates to the neighbors), known as Gather-Apply-Scatter (GAS) model. In the *synchronous* execution mode of PowerGraph, each of these micro-steps is separated by a barrier. The *asynchronous* mode of PowerGraph executes GAS phases without barrier synchronization. However, before each GAS iteration can proceed, active vertices need to acquire locks on their neighbors to prevent two neighbors from choosing the same value simultaneously. Acquiring lock on a high-degree active vertex can limit scalability of an algorithm in PowerGraph for power-law graphs.

Graph Representation. The graph is distributed across different compute nodes using block distribution and is represented with a distributed compressed sparse row (CSR) data structure. For SSSP algorithms, edge weights have been chosen randomly within the range of $[0, 255]$. Execution time of weak scaling plots have been truncated to 1000s. We report

our experimental results as averages of 8 runs for weak scaling results. Since the standard deviation is small, we eliminate error bars in our plots to avoid cluttering.

3.6.2. Weak Scaling Results With RMAT Graphs. For weak scaling experiments, we double the number of compute nodes as we double the number of vertices.

3.6.2.1. *Algorithms with monotonic updates.*

Single-source shortest paths. We report weak scaling results for Δ-stepping, KLA, PowerGraph and our SSSP algorithms with RMAT graph inputs in Figs. 3.3 to 3.6. In general, our algorithm runs 2-3x times faster than other algorithms at larger scales. RMAT-G, Graph500 and RMAT-B have many high-degree vertices, as evident from multiple local maxima in the degree distribution plots [25]. In such cases, while Δ-Stepping and KLA suffer from global synchronization and unbalanced workload, our algorithm benefits from optimistic execution, investing the time gained by eliminating barriers in local ordering. We evaluated different Δ and k values and have set $\Delta = 3$ and $k = 1$ in these experiments, as these values have shown to result in best-performing algorithms with specified weight range. RMAT graphs generally have smaller diameters. Δ-Stepping algorithm in these cases outperform KLA. With larger inputs, PowerGraph asynchronous execution engine has to acquire lock on high-degree vertices, which quickly becomes a bottleneck for performance. Also in many cases, PowerGraph runs out of memory. We also show execution time of PowerGraph with synchronous engine. Although, synchronous engine performs well for smaller scale with Graph500 input, synchronization bottleneck penalizes larger scale performance.

Connected components. Figures 3.7 to 3.10 show the weak scaling results for different connected component algorithms. PowerGraph implements a label-propagation based CC algorithm that generally works well for Graph500 and RMAT-B for smaller inputs. At larger scales, PowerGraph quickly runs out of memory. Since single node experiments were performed with networking turned on, our algorithm runs slower in single-node experiments due to the overhead of scheduling and networking involved with excess work execution. However, our algorithm performs well at larger scales and has better

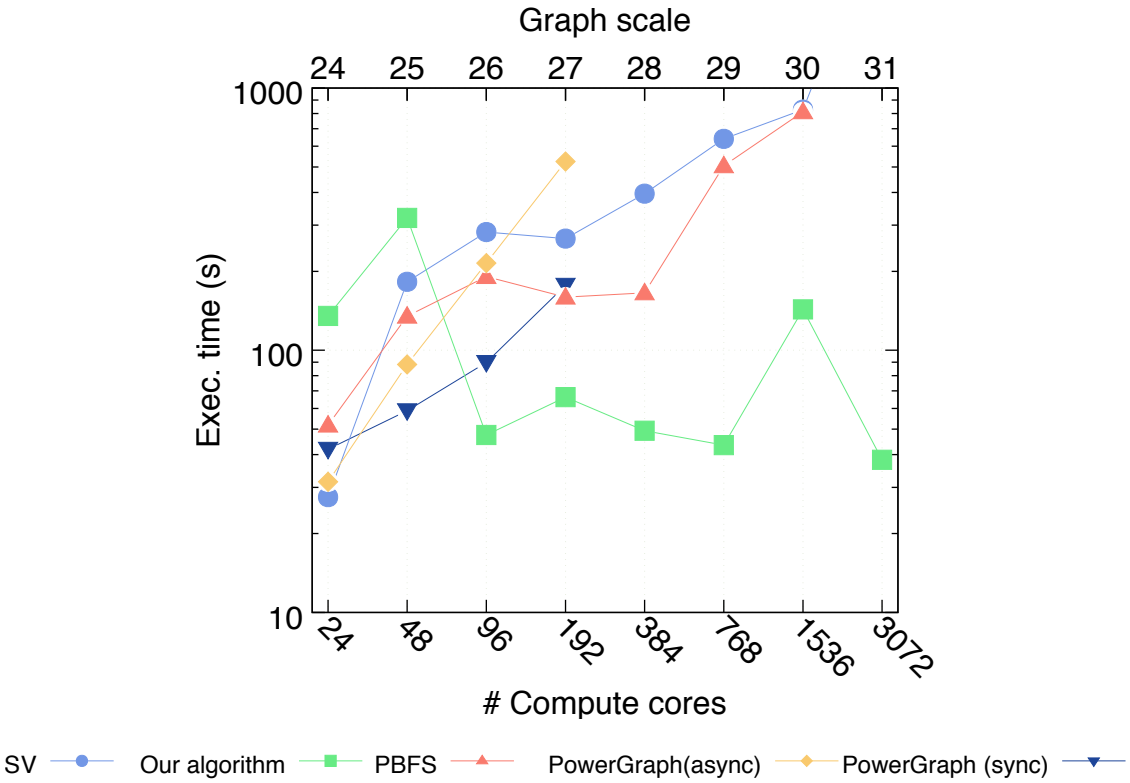


FIGURE 3.7. Weak scaling results for connected component algorithms with RMAT-ER input.

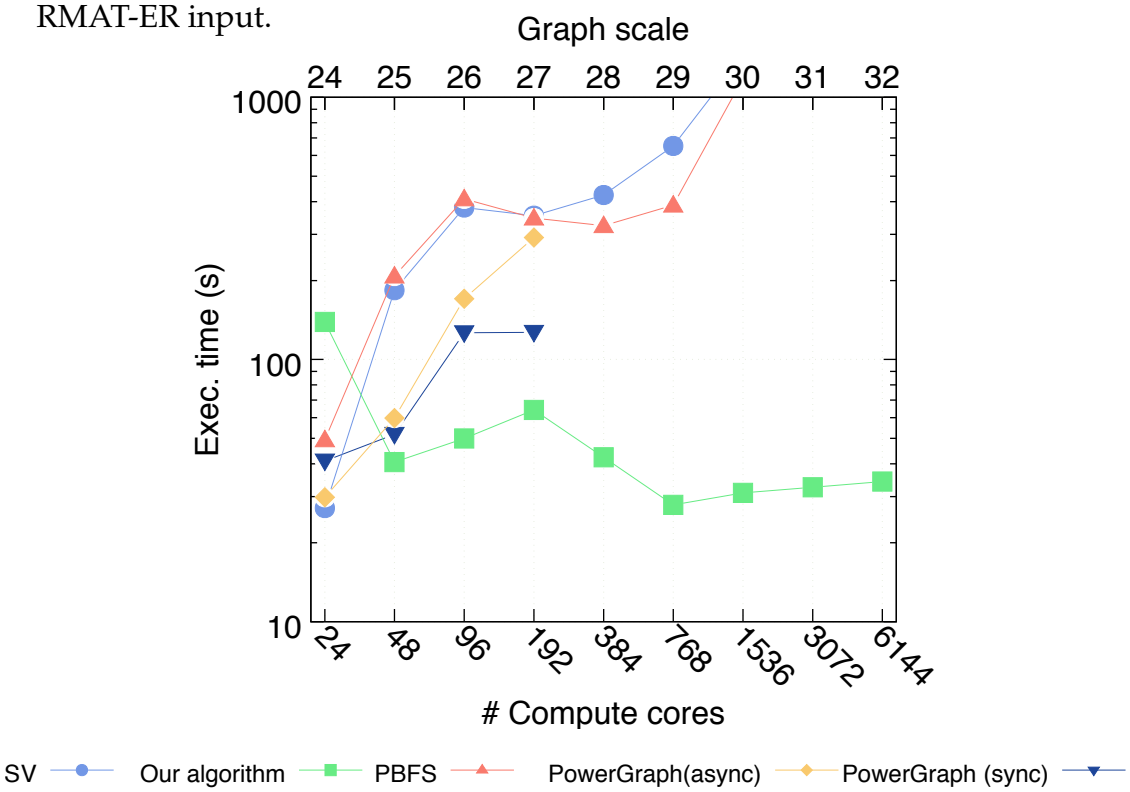


FIGURE 3.8. Weak scaling results for connected component algorithms with RMAT-G input.

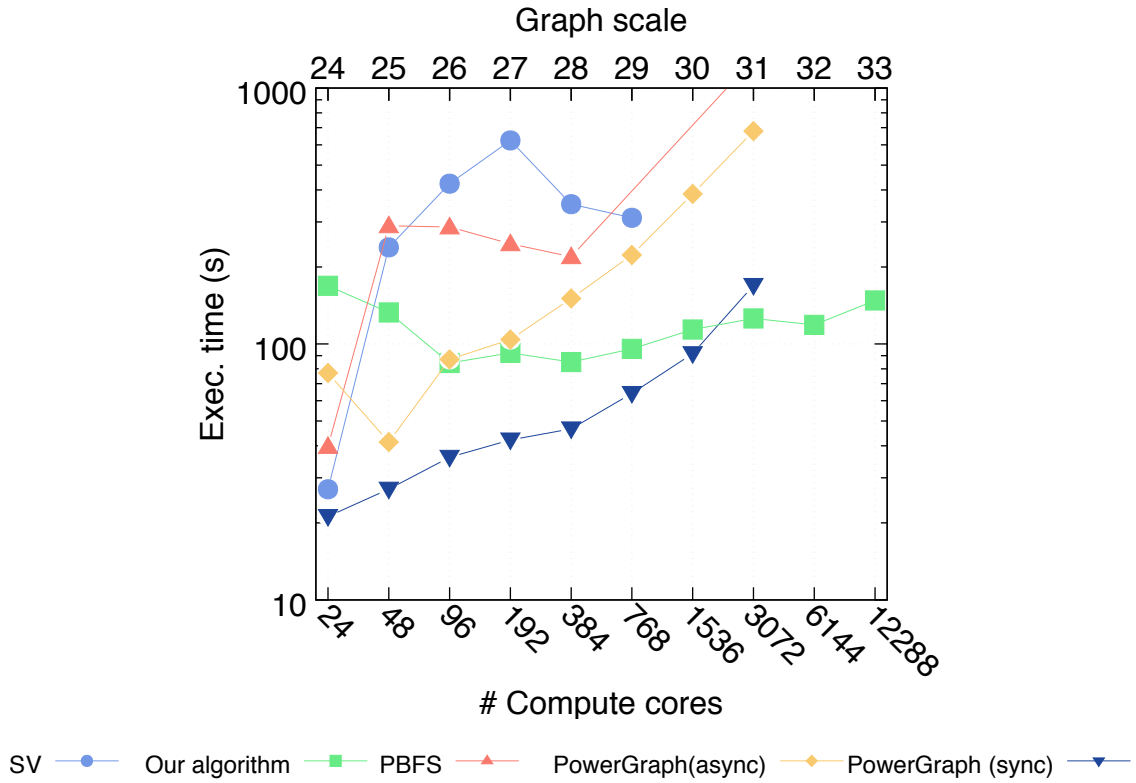


FIGURE 3.9. Weak scaling results for connected component algorithms with Graph500 input.

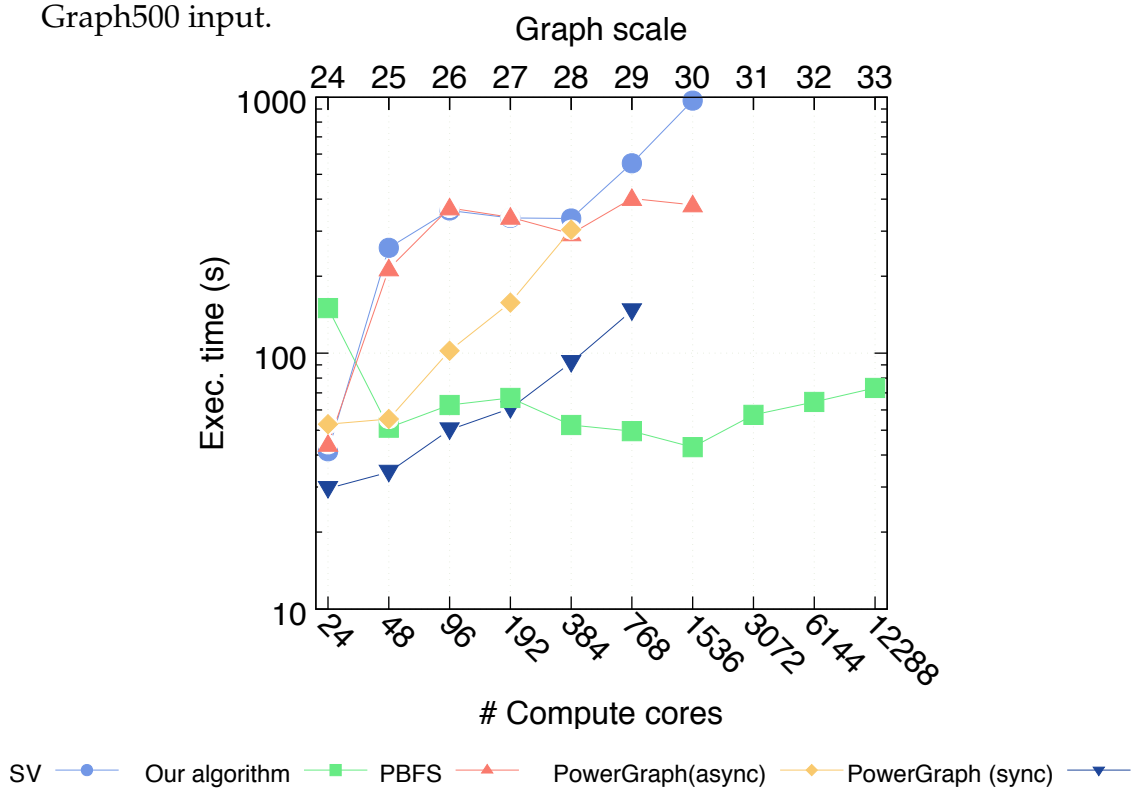


FIGURE 3.10. Weak scaling results for connected component algorithms with RMAT-B input.

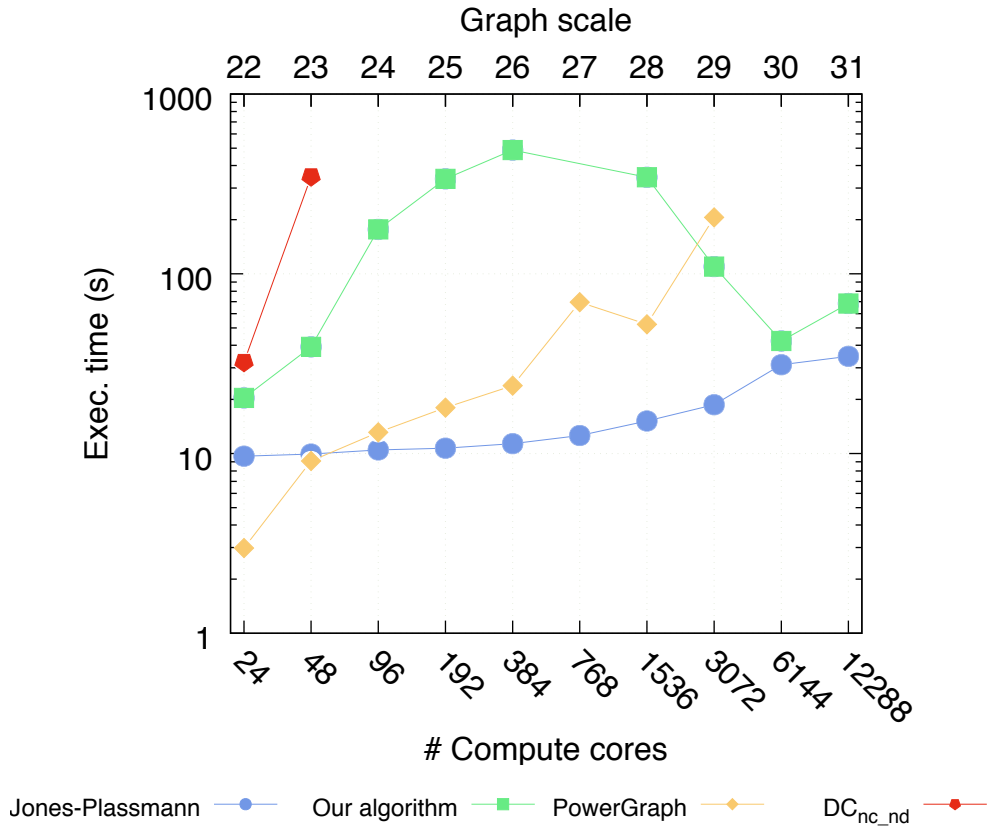
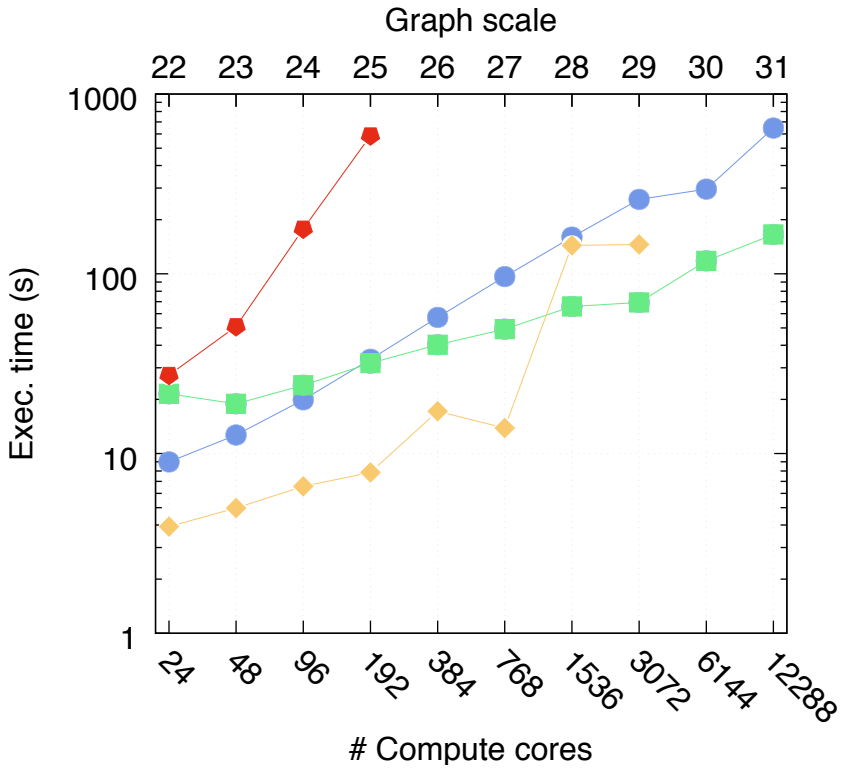


FIGURE 3.11. Weak scaling results for coloring algorithms with RMAT-ER input (color count 16).

scalability compared to other algorithms. PBFS runs faster than SV, since finding largest component first and then starting SV from the root of the component eliminates many intermediate iterations of SV, thus reducing synchronization overhead of hooking and shortcutting phases. PowerGraph's label-propagation algorithm performs better than these two algorithm for skewed graphs with high-degree vertices. This also demonstrates that synchronization overhead can restrict performance with such graph inputs.

3.6.2.2. *Algorithm with non-monotonic updates.*

Graph coloring. Figures 3.11 to 3.14 report the weak scaling results and color qualities for different graph coloring algorithms in AM++ and PowerGraph. We ran simple coloring algorithm with the asynchronous graph processing mode of the PowerGraph framework. With the synchronous execution mode, PowerGraph coloring algorithm fails

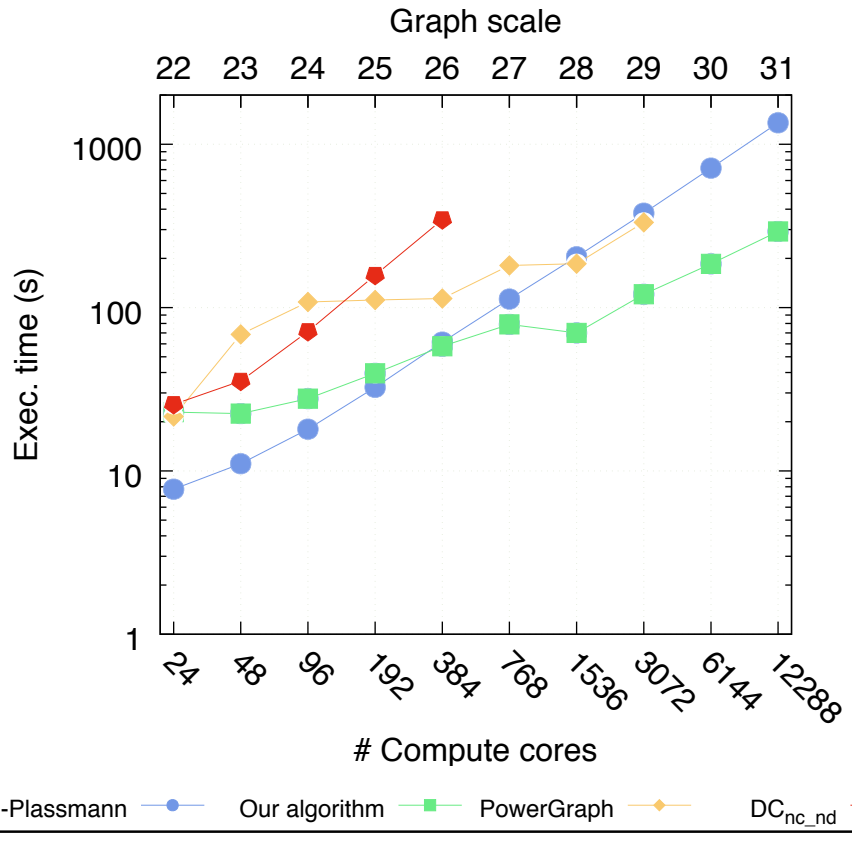


Scale	22	23	24	25	26	27	28	29	30	31
AM++ color count	90	97	106	117	129	140	154	169	185	204
PowerGraph color count	99	111	122	136	148	164	240	214	-	-

FIGURE 3.12. Weak scaling results for coloring algorithms with RMAT-$\widetilde{G}$ input.

to converge [111]. We also tried to run two other vertex-coloring algorithms in PowerGraph with different ordering heuristics: saturation-ordered and degree-ordered coloring. Unfortunately these two algorithms fail to complete execution in a reasonable time.

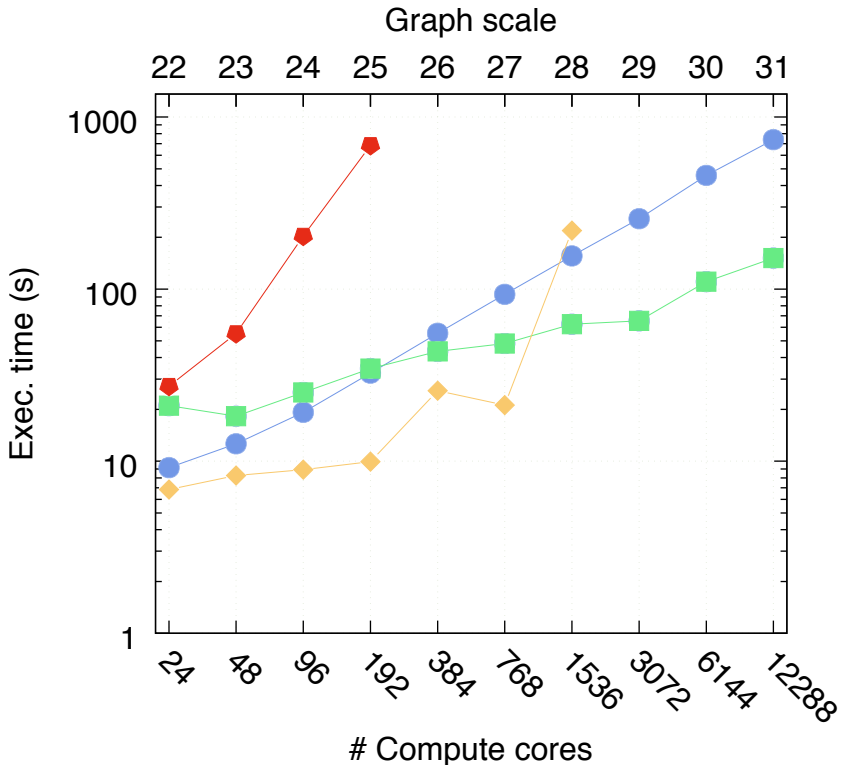
For smaller scale and for graphs with sparsely connected sub-communities (RMAT-ER, RMAT-$\widetilde{G}$) PowerGraph and JP algorithms are faster. With RMAT-ER graph input (Fig. 3.11), Distributed Control does not perform well with smaller node count. With larger node count, *DC* performs comparably with Jones-Plassmann algorithm.



Scale	22	23	24	25	26	27	28	29	30	31
AM++ color count	347	434	530	636	780	955	1159	1413	1714	2074
PowerGraph color count	408	614	776	946	1157	1579	1816	2232	-	-

FIGURE 3.13. Weak scaling results for coloring algorithms with Graph500 input.

However, Graph500 (Fig. 3.13), RMAT-B (Fig. 3.14), and large-scale RMAT-$\widetilde{G}$ (Fig. 3.12) graph structures have denser connections among sub-communities and require more colors (color counts are tabulated under the plots). Larger parameter values for a and d, in comparison to b and c values, generate sub-communities within the graph structures. The vertices within these local sub-graphs are highly connected and forms almost cliques. The larger and wider range of values for clustering coefficient of RMAT-$\widetilde{G}$ and RMAT-B [25] graphs also validate the existence of dense sub-communities. With a pre-execution ordering heuristic for vertex-coloring, each vertex in such dense local sub-graph has to wait for its predecessors before obtaining a color. As a result, with larger number of dense



Scale	22	23	24	25	26	27	28	29	30	31
AM++ color count	169	192	223	256	295	342	399	454	526	612
PowerGraph color count	207	242	280	330	395	383	617	-	-	

FIGURE 3.14. Weak scaling results for coloring algorithms with RMAT-B input.

local sub-graphs, parallelization in Jones-Plassmann becomes limited. Such requirements result in JP and PowerGraph algorithms to perform worse. In these graphs, vertices have large number of predecessors and vertex-centric barriers implicitly imposed by JP and PowerGraph prevent vertices from obtaining colors till all their predecessors have been colored. Consequently, the waiting time becomes a bottleneck for performance. On the contrary, our algorithm can speculatively proceed forward, optimistically propagating color information without waiting on vertex-centric barriers. In such cases, our algorithm outperforms other implementations with a speedup of 2-3x.

With RMAT-ER graph (Fig. 3.11), *DC* can suffer from performance bottleneck if too many sub-optimal updates are performed. This is evident from the workload characteristics of *DC* with RMAT-ER, shown in Fig. 3.30. *DC*, in this case, is unable to successfully filter out sub-optimal work and suffers from extra work execution. In particular, with smaller compute node count, *DC* suffers from performance bottleneck due to the overhead encountered by execution time ordering and frequent conflicts that arise from optimistic color update. Frequent color update makes *DC* a compute-intensive algorithm, rather than a communication-bound algorithm. However, the situation reverses at scale and *DC* catches up with Jones-Plassmann at scale 30. We will discuss more about workload characteristics of the two algorithms in Sec. 3.6.6.

The color qualities have also been tabulated in each case. With increasing power-law characteristics, the number of colors required also increases. However, both JP and DC coloring algorithms achieve the same coloring quality. The requirement for more colors with the increase of power-law characteristics can be attributed to the dense local sub-graphs (sub-communities).

3.6.2.3. *Effect of Caching.* We also show scaling results of *DC* coloring with caching and priority heuristic disabled (DC_{nc_nd}) in Figs. 3.11 to 3.14. As can be seen from the figure, even at small scale, DC_{nc_nd} does not perform well due to work explosion resulting from aggressive speculation. At or beyond 384 compute cores, the amount of network traffic generated by DC_{nc_nd} causes node failures due to memory exhaustion.

3.6.3. Results With Real-world Dataset.

3.6.3.1. *Algorithms with monotonic updates.*

Single-source shortest path. Table 1 tabulates speedups of our algorithm over baseline algorithms and PowerGraph. We also include which baseline algorithm (Δ-stepping or KLA) is better in each case. Graphs with longer diameters such as road networks perform well with KLA algorithm. Our algorithms outperform both baseline and PowerGraph implementation in most cases. With smaller road networks, the average degree of each

Graphtype	Graph	|V|	|E|	$\tilde{D}$	cf	S_{sssp_am}	S_{sssp_pg}	S_{cc_am}	S_{cc_pg}	$S_{coloring\_am}$	$S_{coloring\_pg}$	Color count
Communication networks	wiki-Talk	2.4M	5M	9	0.0526	1.48(Δ)	1.29	4.01	2.76	0.4	1.93	79
	email-EuAll	265k	420k	14	0.0671	2.8(KLA)	3.6	3.71	6.47	0.4	4.15	30
Social networks	Friendster	65M	3.6B	32	0.1623	5.47(KLA)	-	13.4	-	1.47	-	155
	Twitter	44M	2.9B	36	0.0846	1.74(Δ)	-	3	-	0.85	-	1084
	soc-LiveJ.	4.8M	69M	16	0.2742	2.05(Δ)	1.74	9.3	12.08	0.76	1.87	324
	com-orkut	3M	117M	9	0.1666	3.15(Δ)	-	4.17	4.4	1.12	0.83	115
	com-lj	4M	34M	17	0.28	3.09(Δ)	2.08	7.78	9.24	0.78	1.3	333
	com-youtube	1.1M	2.9M	20	0.0808	2.9(KLA)	-	5.17	4.01	0.42	2.04	38
	com-dblp	317k	1M	21	0.6324	1.81(KLA)	2.36	4.18	7.72	0.53	3.43	113
	com-amazon	334k	925k	44	0.3967	1.73(KLA)	-	3.68	8.63	0.43	1.68	9
Purchase network	amazon0601	403k	3.3M	21	0.4177	2.48(Δ)	2.76	4.85	9.6	0.53	1.4	12
Road networks	roadNet-CA	1.9M	5.5M	849	0.0464	2.86(KLA)	2.36	1.06	15	0.66	22	4
	roadNet-TX	1.3M	3.8M	1054	0.0470	7.09(KLA)	6.01	0.82	11	0.5	24	4
	roadNet-PA	1M	3M	786	0.0465	4.478(KLA)	4.4	0.85	9.96	1	19	4
	europe_osm	50M	1B	$\sim$ 7000	-	4.76(KLA)	-	20	-	0.34	-	4
Citation graphs	cit-Patents	3.7M	16.5M	22	0.0757	4.4(Δ)	-	9.9	11	0.33	1.2	14
Web graphs	Web-Google	875k	5.1M	21	0.5143	4(KLA)	3.1	5.6	8.78	0.5	1.5	43
	Web-BerkStan	685k	7.6M	514	0.5967	5.06(KLA)	-	4.2	14	1.2	6.3	201
	Web-Stanford	281k	2.3M	674	0.5976	6.07(KLA)	-	3.6	7.7	0.7	2.1	63
	sk-2005	50M	3.8B	17	0.23	3.57(KLA)	-	7.4	-	4.81	-	4511

TABLE 1. Speedup results of our algorithms with real-world input graphs. Total number of vertices ($|V|$), edges ($|E|$),diameter ($\tilde{D}$), average clustering coefficient (cf) for each graph input is tabulated here. S_{sssp_am} denotes speedups of our SSSP algorithm over best baseline algorithm (mentioned in each case). S_{sssp_pg}, S_{cc_pg} and $S_{coloring\_pg}$ denote speedups of our SSSP, CC and coloring algorithms over PowerGraph implementations respectively. Since, with all real-world inputs PBFS outperforms SV, S_{cc_am} denotes speedups of our CC algorithm over PBFS. Speedup of our coloring algorithm over JP is denoted by $S_{coloring\_am}$. Both JP and DC algorithms achieve same color quality. We report color quality of our algorithm in each case, which is better than Powergraph. Powergraph fails to execute with some inputs.

.

vertex is 4 and there is not much parallelism available for our algorithm to explore optimistic execution. In such cases, our algorithm performs slower than PowerGraph due to sub-optimal work execution. We have also run our algorithms with larger inputs such as Friendster, Twitter and europe_osm and got speedups of 5.47, 1.74, and 4.76 respectively over best-performing baseline algorithms.

Connected components. We also report speedups of our algorithm over baseline algorithms and PowerGraph in Table 1. In all reported cases, our algorithm outperforms other algorithms. With larger inputs, such as Friendster, our algorithm runs 13 times faster compared to best baseline algorithm.

3.6.3.2. *Algorithm with non-monotonic updates.*

Graph coloring. Table 1 reports speedup of our algorithm over PowerGraph as well as the color quality of our algorithm. Our algorithm outperforms PowerGraph with most inputs. Notably, these real world datasets do not require many colors. Hence, we do not observe any performance benefit from speculative execution of our algorithm over Jones-Plassmann with such graphs. However, with larger graph inputs with densely connected sub-communities (for example sk-2005, a collaboration network among Slovakian scholars), we have observed significant performance gain. We report our experimental results with such larger real-world graph in Table 1. Road networks have limited opportunity for optimistic execution. Twitter dataset we have used here is a follower network, where many people can follow a celebrity, but its highly unlikely for the followers to be connected with each other. Thus they do not form any connections among sub-communities. In both cases, our algorithm performs slower for the reasons stated.

3.6.4. Strong Scaling Results. For strong scaling experiments, we double the number of compute nodes and keep the graph size constant.

3.6.4.1. *Algorithms with monotonic updates.*

Figures 3.15 to 3.17 report strong scaling results for different SSSP algorithms with large real-world datasets as well as synthetic graphs. With increasing cores, DC shows better scalability in general. However, beyond certain amount of cores, runtime scheduling and

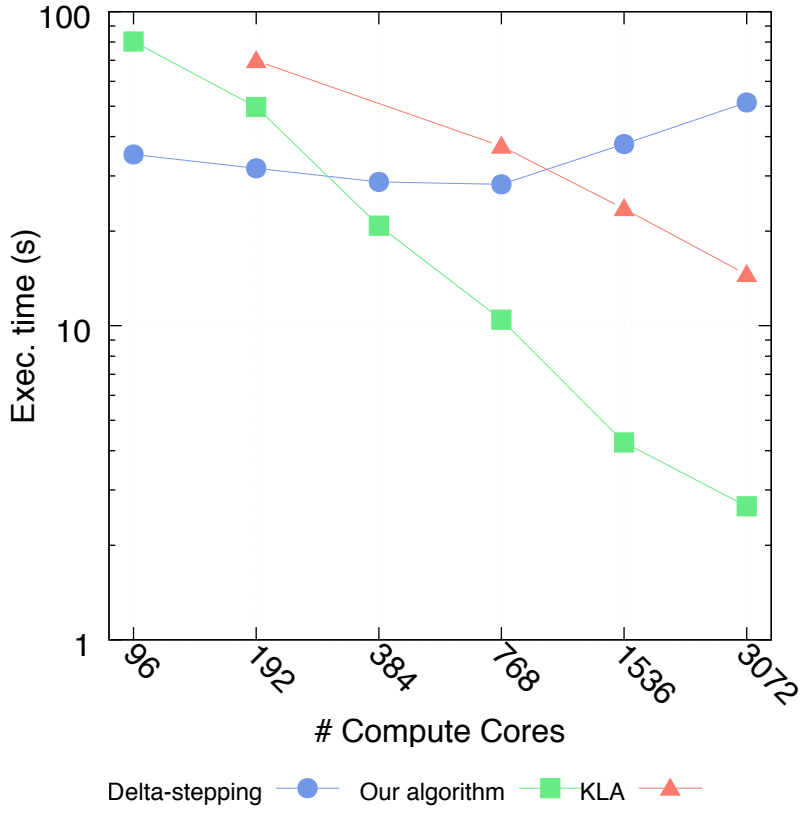


FIGURE 3.15. Strong scaling results for SSSP algorithms with Friendster input.

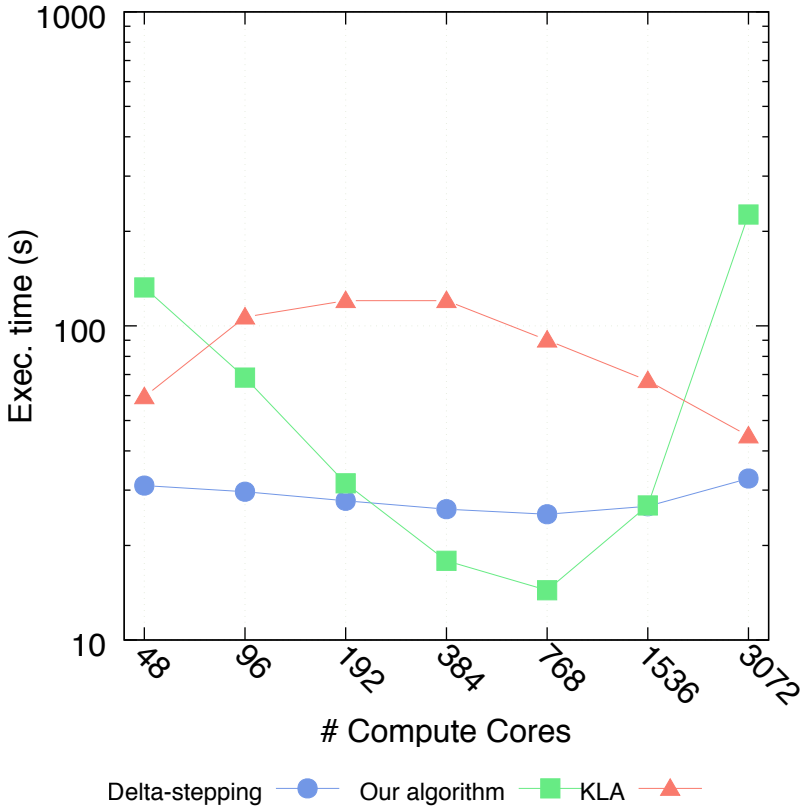


FIGURE 3.16. Strong scaling results for SSSP algorithms with Twitter input.

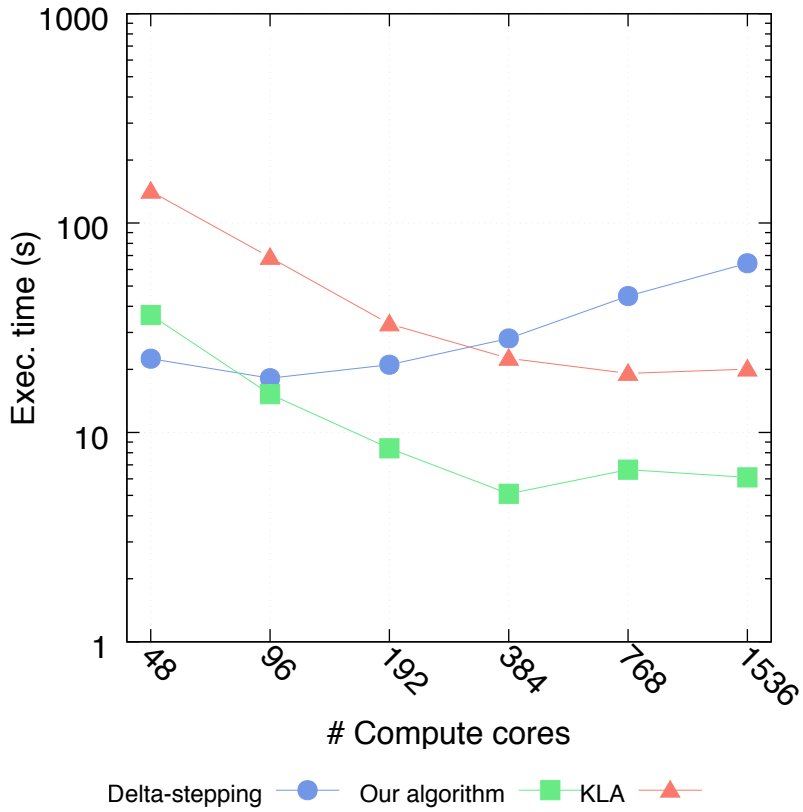


FIGURE 3.17. Strong scaling results for SSSP algorithms with sk2005 input.

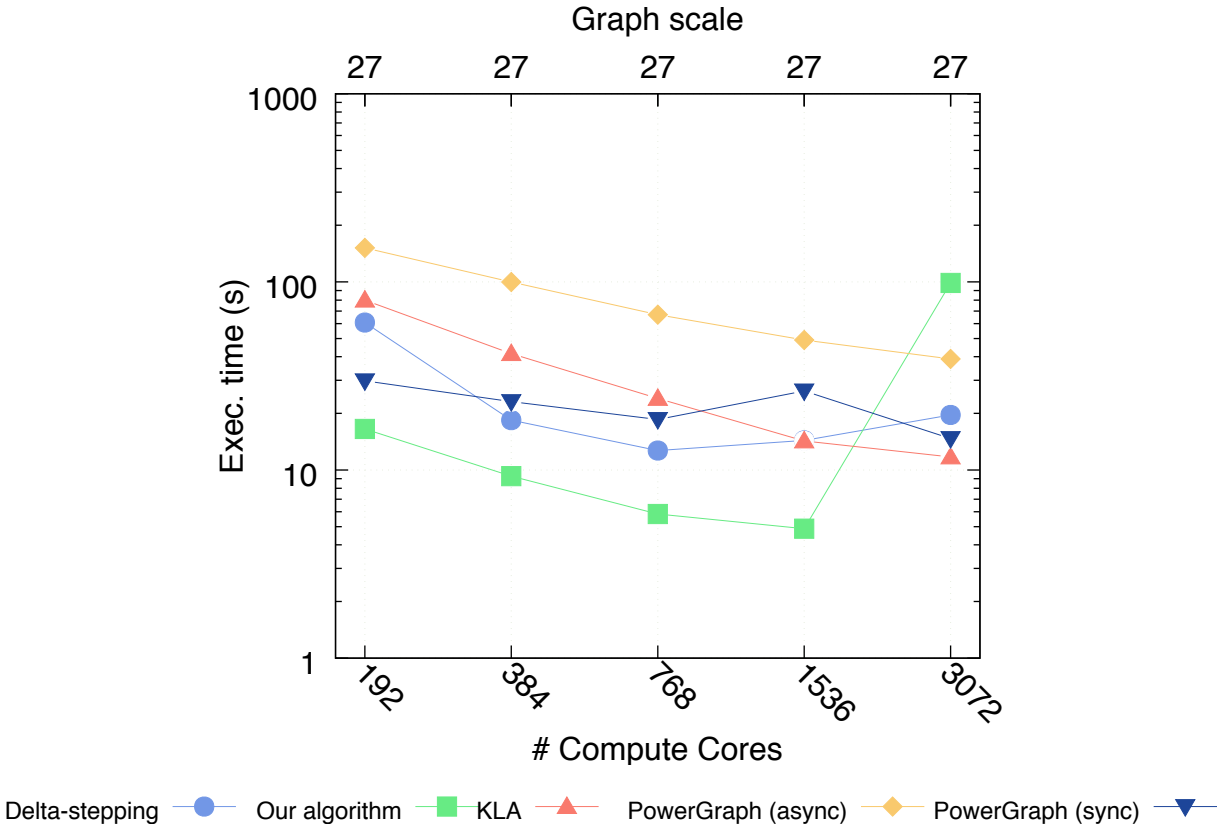


FIGURE 3.18. Strong scaling results for SSSP algorithms with Graph500 input.

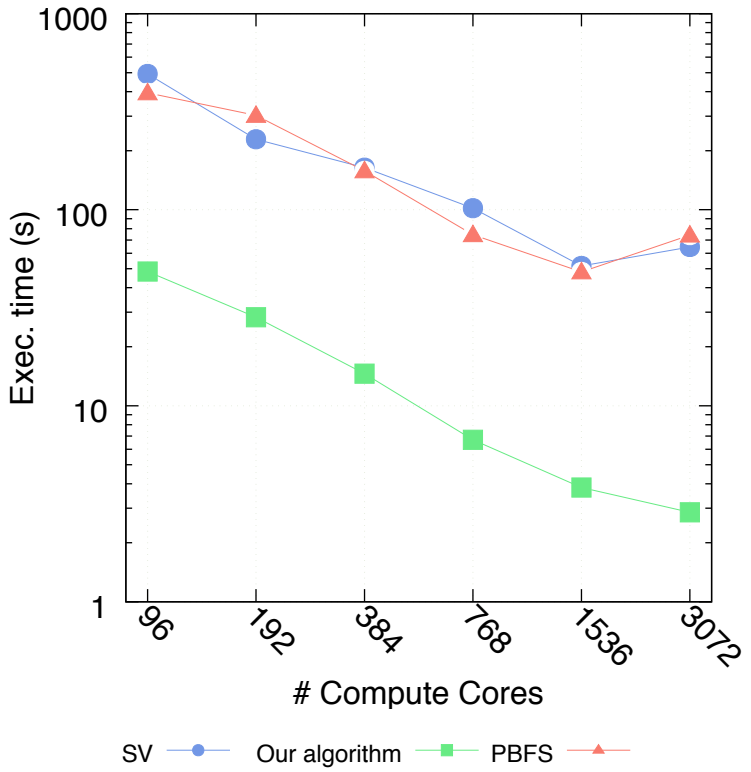


FIGURE 3.19. Strong scaling results for connected component algorithms with Friendster input.

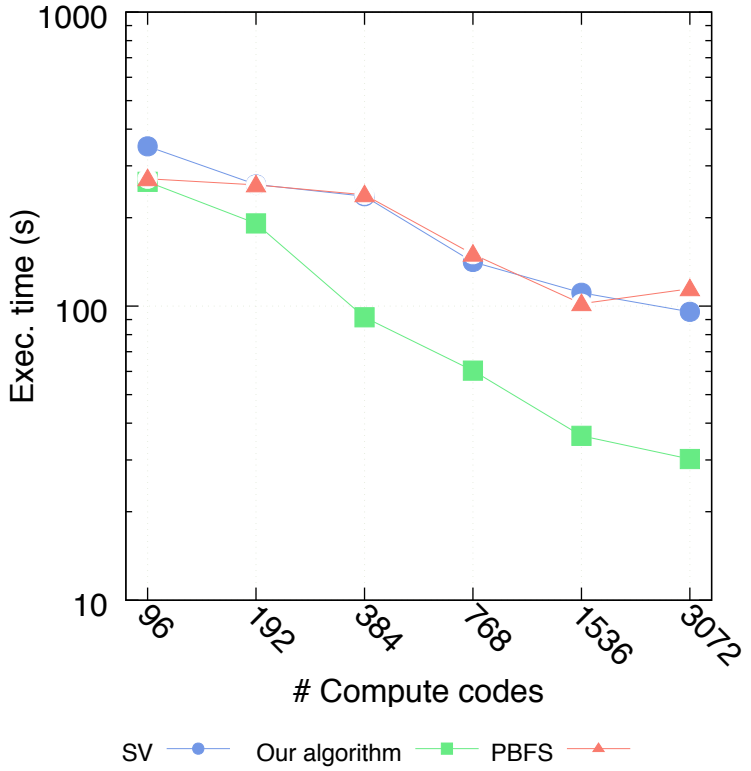


FIGURE 3.20. Strong scaling results for connected component algorithms with Twitter input.

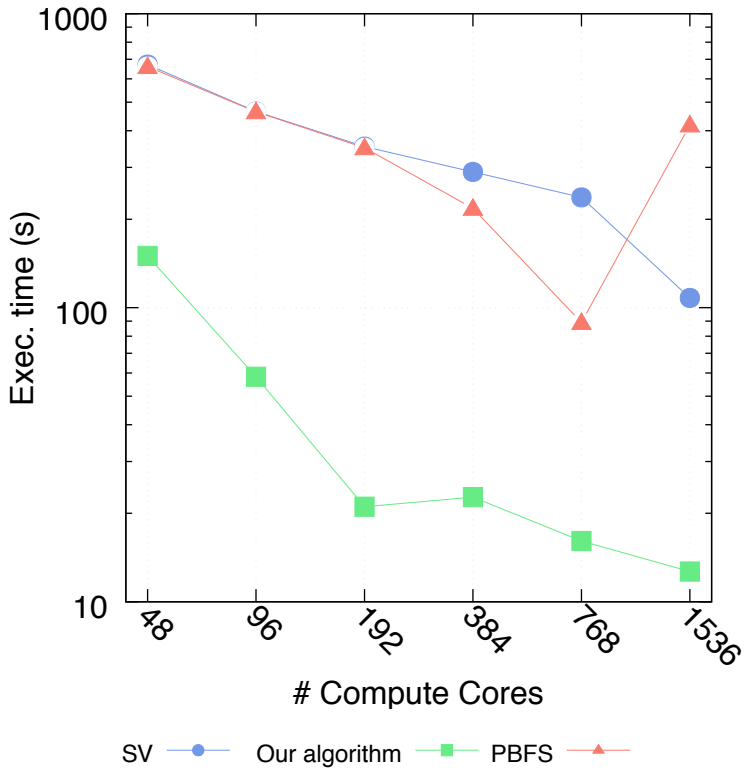


FIGURE 3.21. Strong scaling results for connected component algorithms with sk2005 input.

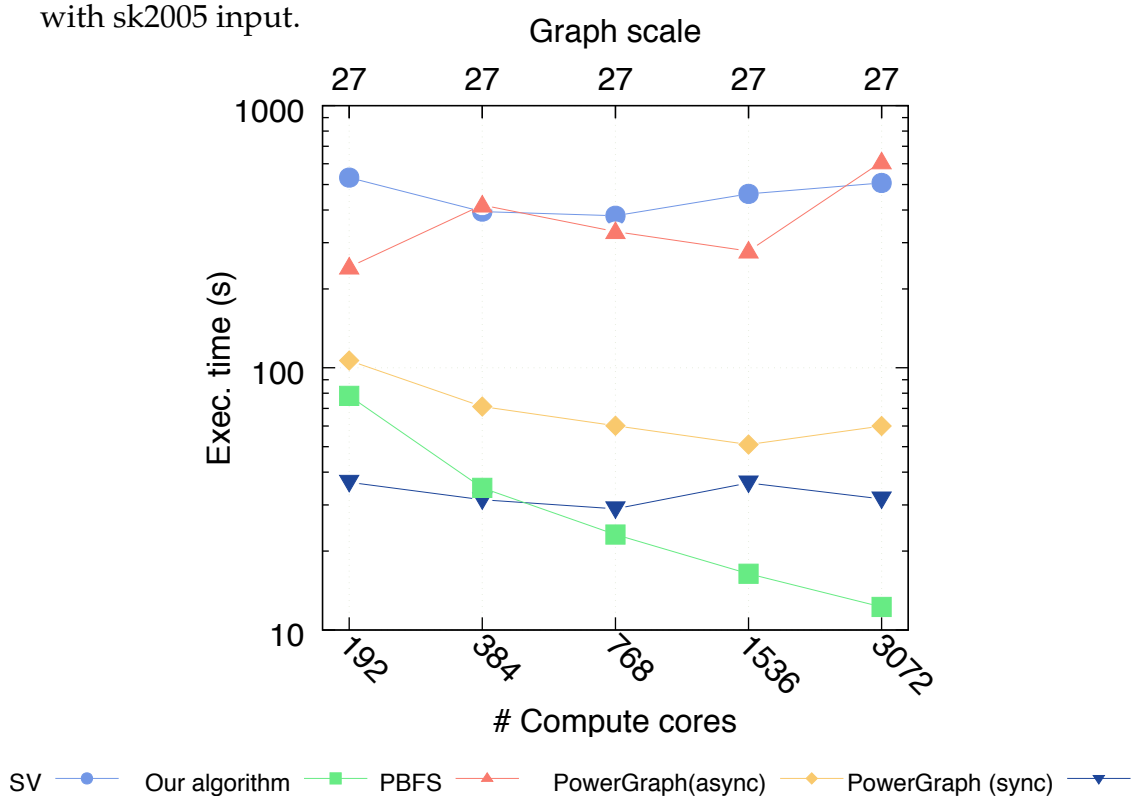


FIGURE 3.22. Strong scaling results for connected component algorithms with Graph500 input.

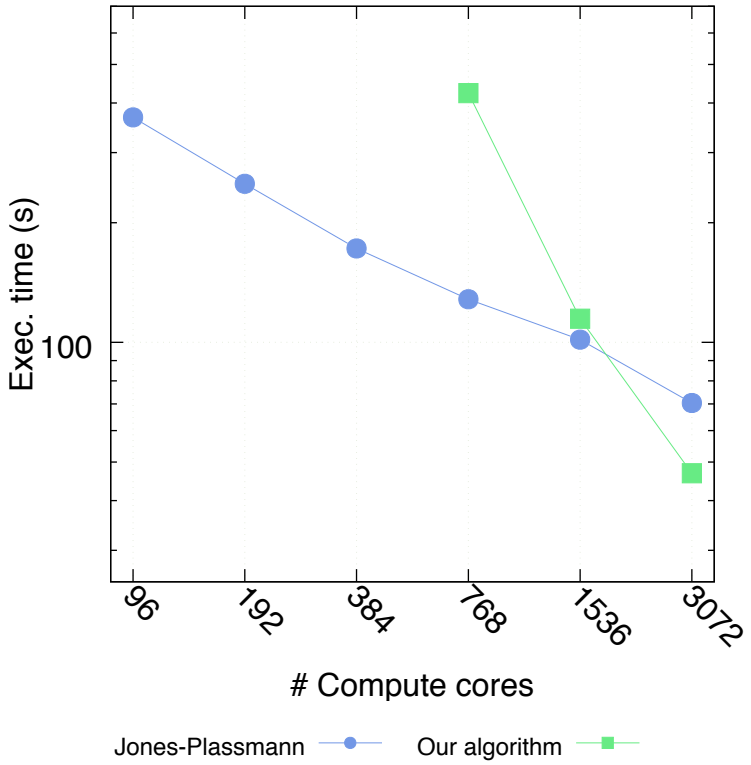


FIGURE 3.23. Strong scaling results for coloring algorithms with Friendster input (155 colors).

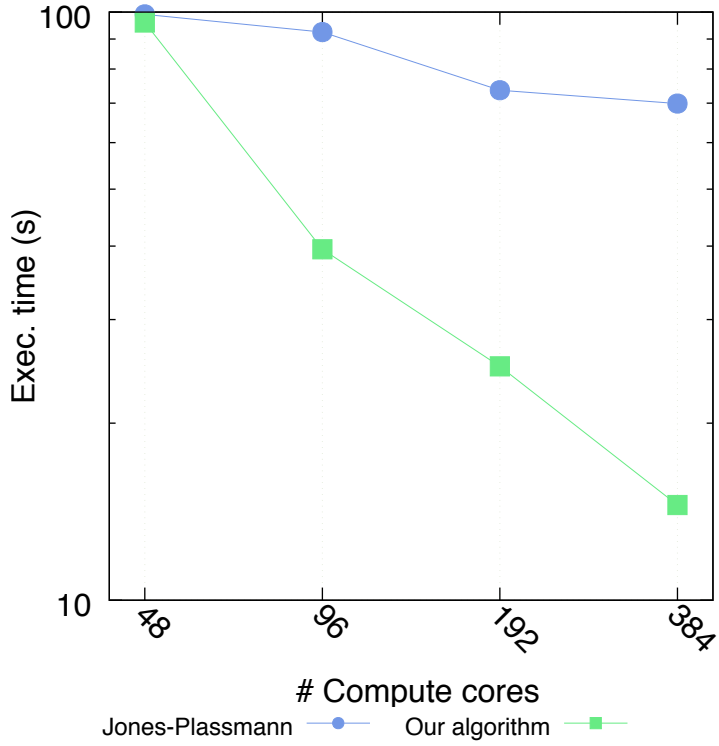


FIGURE 3.24. Strong scaling results for coloring algorithms with sk2005 input (4511 colors).

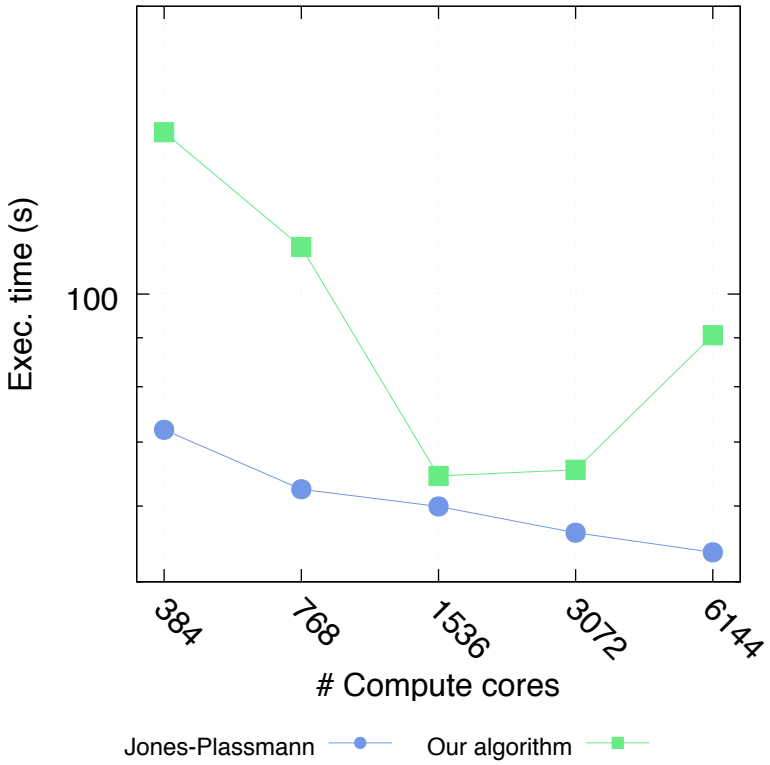


FIGURE 3.25. Strong scaling results for coloring algorithms with Twitter input (1084 colors).

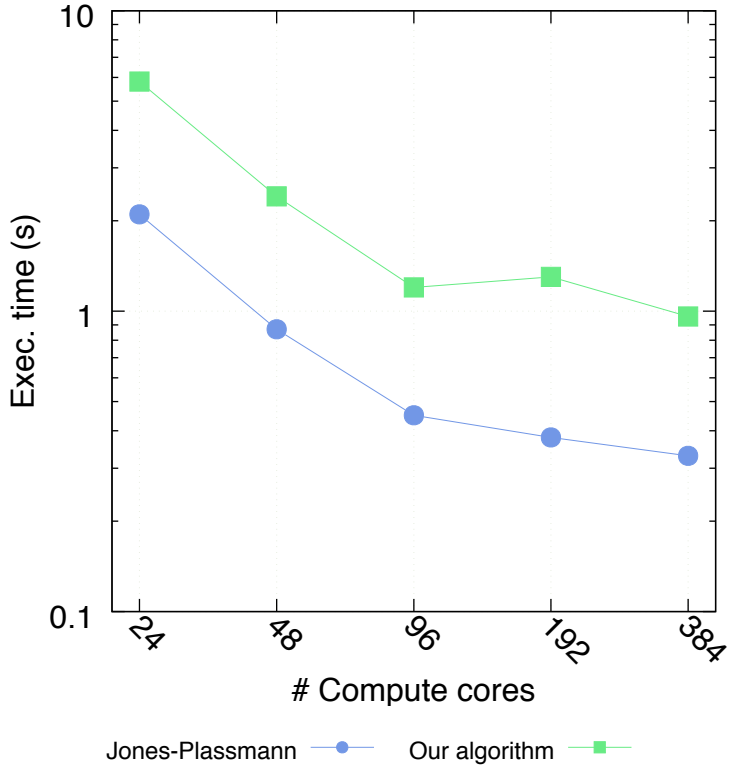


FIGURE 3.26. Strong scaling results for coloring algorithms with europe-osm input (4 colors).

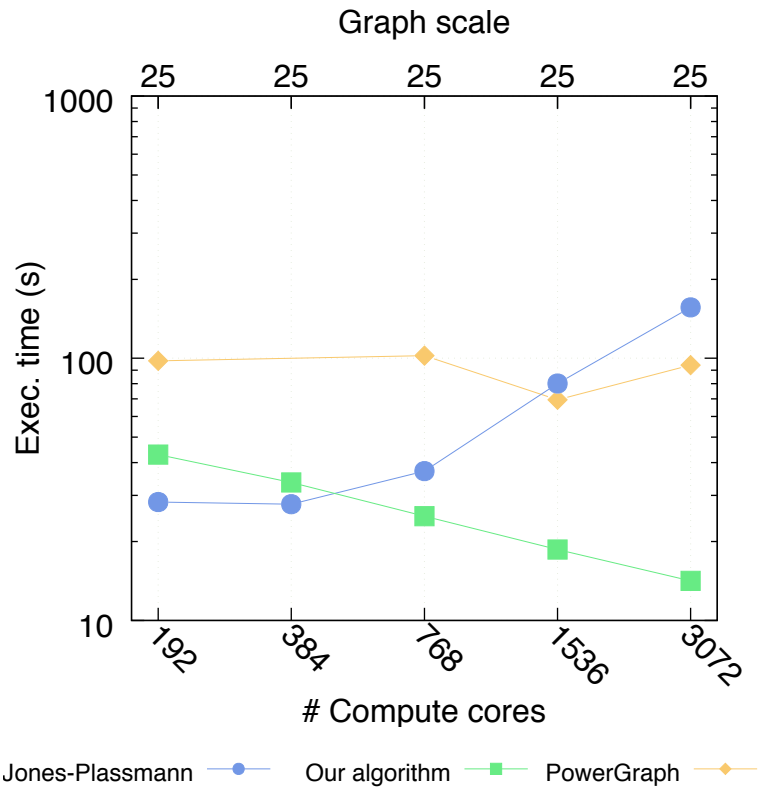


FIGURE 3.27. Strong scaling results for coloring algorithms with Graph500 input (636 colors).

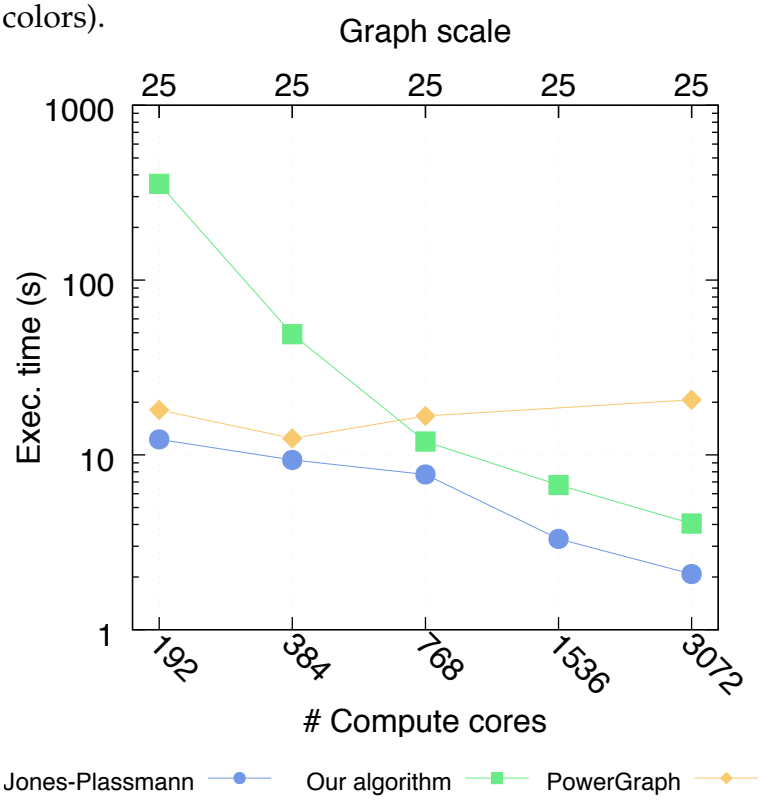


FIGURE 3.28. Strong scaling results for coloring algorithms with RMAT-ER input (15 colors).

communication overhead of asynchronous execution factors in and penalizes performance with Twitter and Graph500 datasets. For connected component algorithm, however, DC shows better scalability with all datasets (Figs. 3.19 to 3.22).

3.6.4.2. *Algorithms with non-monotonic updates.*

Figures 3.27 and 3.28 report strong scaling results for different coloring algorithms with RMAT graphs. With Graph500 input (Fig. 3.27), *DC* achieves better performance with higher number of cores. JP, on the other hand, starts to suffer from vertex centric barriers with larger cores due to the distribution of graphs. As we increase the number of node counts, Jones-Plassmann algorithm struggles to scale, since vertex-centric barrier becomes an issue and communication overhead across large number of compute nodes starts effecting its performance. *DC*, on the other hand, enjoys the opportunity of optimistic parallelism with larger resource count. With enough processing units at its disposal, *DC* can support continuous color updates and saturates the computing resources with work. In this way, even though *DC* has to execute more work compared to JP (Sec. 3.6.6), investing the time obtained by eliminating vertex-centric barrier, results in better performance.

We also evaluate strong scaling of coloring algorithms on AM++ with four larger real-world datasets: Friendster, Twitter, sk-2005 and europe_osm (Figs. 3.23 to 3.26). These datasets represent social network, webcrawl graphs, follower network and road networks respectively. With Friendster (Fig. 3.23) and sk-2005 input (Fig. 3.24), *DC* has better scalability compared to JP. In both cases, increasing the number of compute nodes penalizes JP with communication cost as well as vertex-centric barriers. Note that, sk2005 requires large number of colors compared to other input graphs. sk2005 represents collaboration network of Slovakian researchers, thus represent communities that are densely connected (hence larger color count). Web-crawl graphs (such as sk2005) have two important topological characteristics: they have low diameters ("small world") and their degree distribution follow power-law ("scale-free"). Road network graphs such as europe_osm, on the other hand, has bounded degree distribution, a smaller average and maximum degree count but very high diameter (cf. Table 1). Since there is not much scope for optimistic parallelism

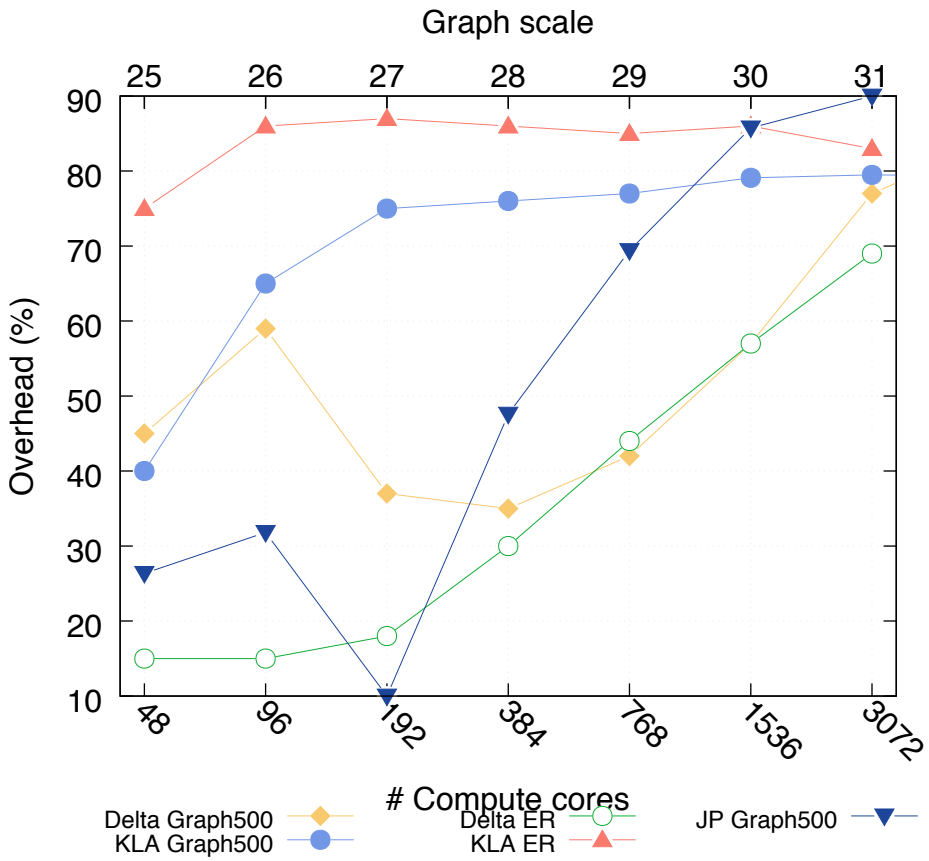


FIGURE 3.29. Barrier overhead.

due to low degree-count, *DC* has slower execution time than JP in this case(Fig. 3.26). With Twitter input, *DC* achieves comparable performance with 1536 cores (Fig. 3.25). The particular Twitter dataset we have experimented with is generated from the follower network. This social network graph also has a very small clustering coefficient (0.0846%) , suggesting that followers are not connected to each other and hence do not create dense local subgraphs. This is in contrast to the topological structure we observe in RMAT-B, RMAT-$\widetilde{G}$ and Graph500 inputs, which have multiple dense locally connected subgraphs.

3.6.5. Synchronization Overhead. We measure the average global barrier synchronization overhead for Δ-stepping and KLA and vertex-centric barrier overhead for JP and report the results in Fig. 3.29. Such overhead captures per-task execution delay due to barriers. Performance difference between Δ-stepping and KLA (in Figs. 3.3 to 3.6) can

be explained from the overhead calculation. On average, KLA incurs more overhead compared to $\Delta-$stepping.

We measure the overhead of vertex-centric barriers in Jones-Plassmann algorithm as follows: for each vertex, we record the time when the first message from a predecessor is received. Next, we record the time of the receipt of last predecessor update. The difference between these two quantities gives a measure of how much time each vertex wait on a barrier. We compute the average of such time with different RMAT inputs and scales and report the result.

3.6.6. Workload Characteristics. Figure 3.30 shows the breakdown of different types of work executed by each vertex-coloring algorithm. We classify workload performed by each algorithm in 3 categories: useful,useless, and rejected work. *Useful* works are the aggregate count of tasks that contain the final predecessor color values and result in successful color updates. Both Jones-Plassmann and Distributed Control execute the same amount of such work over the course of execution. Useful work yields final vertex colors. The other two types of tasks are specific to the *DC* coloring algorithm. In *DC*, whenever a better color value becomes available, it forces an invalidation of the current color and ultimately triggers a correction of the current vertex color value. These updates happen when *DC* processes workitems from the priority queues and tries to update a vertex color with a newly available color. Once updated, the new color information is sent to all the successors. However, instead of sending these updates immediately, we cache these messages for some time. *Useless* work arises when a color update becomes stale in the application-level message cache while waiting to be sent to the successors. *Rejected* work goes over the network but on arrival gets rejected due to containing outdated update. As can be seen in Fig. 3.30, *DC* performs more work compared to JP. However, only rejected work results in network messages. Compared to other inputs, RMAT-ER results in the largest amount of this type of work, hence *DC* has worst performance with RMAT-ER input. Nonetheless, the elimination of vertex-centric barriers in *DC* results in

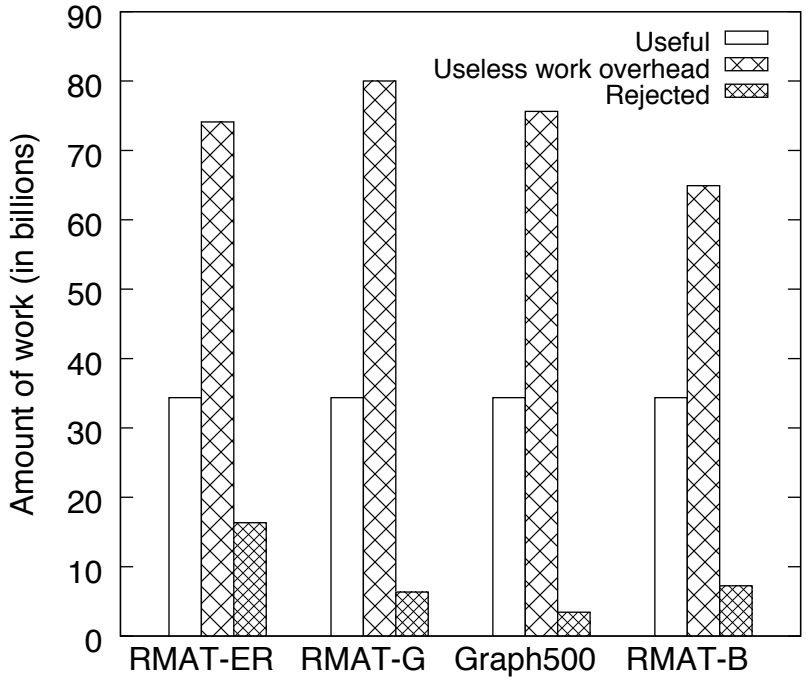


FIGURE 3.30. Workload statistics on 512 compute nodes with scale 31 graph.

Ordering	No. colors	Ordering time (s)	Coloring time (s)	Color ratio
Random	500	0.21	11.68	1.06
Largest first	388	0.2	10.91	1.36
Incidence degree	365	30.49	11.47	1.452
Smallest last	366	32.21	11.36	1.448

TABLE 2. Color quality, sequential ordering time, and coloring time of a Scale 24 Graph500 graph in ColPack [54]. The color ratio column reports ratio of no. of colors obtained by our algorithm to the sequential version with a particular ordering. The ordering is tabulated from the least restrictive ordering to the most restrictive one.

significant performance benefit on graphs with densely connected sub-communities and strong power-law.

3.6.7. Coloring Quality. In Table 2, we compare the color quality of our algorithm (with random ordering heuristic for pre-execution order) to those in a well-known coloring

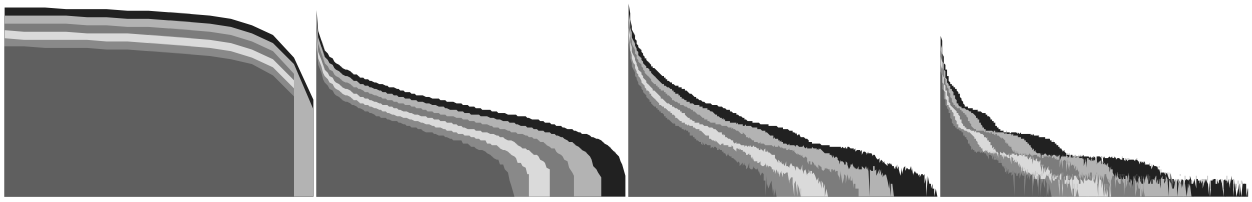


FIGURE 3.31. From left-to-right: Distribution of vertices in different color bins of RMAT-ER, RMAT-$\widetilde{G}$, RMAT-B and Graph500 respectively for weak scaling results.

software package ColPack [54]. ColPack provides sequential and parallel versions of algorithms for a range of graph coloring problems. As can be seen from Table 2, in the worst case, our algorithm uses 1.45 times more colors than the sequential version with incidence-degree ordering. However, in sequential setting, even with small scale 24 graph, it takes 30.49s to obtain the incidence degree order. Smallest last, the most restrictive ordering among the ordering schemes, takes about 32.21s to order the vertices. On the other hand, random ordering does not require any ordering time, since it rely upon the natural representation of vertices in the graph to impose ordering and uses few extra colors. Imposing other ordering in distributed setting is more involved.

3.6.8. Skewness of Sizes of Color Classes. In Fig. 3.31, we show the sizes of different color classes (independent subsets) for 4 types of RMAT graphs. Sizes of color classes can assist to choose between JP and DC. If the sizes of the color classes are such that most of the vertices are concentrated in the lower-numbered color bins (for example in RMAT-B, RMAT-$\widetilde{G}$ and Graph500 in Fig. 3.31), it can be a good predictor of better performance of DC over JP with such inputs. This is because, due to the speculative nature of DC, it tends to choose first the minimum available color in a search and propagates such speculated color choice as soon as possible. The larger the lower color bins are, right predictions will be made earlier in the computation of DC. Moreover, amount of rejected work is highest (Fig. 3.30) with graph that has the most equal color distribution (RMAT-ER).

3.6.9. Comparison With Other Graph Frameworks. We also compare our SSSP and connected component algorithms with two other state-of-the-art frameworks, namely Gemini [115] and Galois [91].

3.6.9.1. *Gemini.* Gemini is a distributed graph processing framework that focuses on efficient partitioning of graph inputs as well as suitable graph representation technique to improve the performance of graph algorithms. This framework employs chunk-based partitioning to distribute the graph while preserving locality by maintaining shadow copies of vertices in a subgraph. Using this technique, Gemini tries to minimize communication at the cost of higher memory requirement. In addition, Gemini applies NUMA-aware sub-partitioning across multiple sockets in a compute node to handle intra-node imbalance. For representing the graph efficiently, Gemini stores incoming edges into Compressed Sparse Column (CSC) and outgoing edges in Compressed Sparse Row (CSR) format. For sparse mode edges, bitmap is used to mark existance of an edge. For dense mode edges, a doubly compression scheme is used for compact representation. Gemini runtime has a work-stealing task scheduler for load balancing and intra-node edge processing.

We compare the performance of our approach for SSSP problem with Gemini with Friendster, Twitter and sk2005 datasets. We report the results in Figs. 3.32 to 3.34. With both Twitter and sk2005 datasets, our algorithm runs slower compared to Gemini. With Friendster input, however, our algorithm is faster at larger scale. We also compare our connected component algorithm with Gemini with these three datasets and observed similar pattern. Here, our algorithm is faster with Friendster input (Fig. 3.36) but slower with Twitter and sk2005 inputs (Figs. 3.35 and 3.37).

However, Gemini spends a huge amount time to pre-process the graph, striving to find a good inter- and intra-node partition to handle load imbalance. We report such preprocessing time also in Figs. 3.32 to 3.37. Combining such pre-processing time with actual execution time in each case results in a total runtime which is slower than our algorithms with all datasets. It has been discussed in the literature that finding a generic partitioner for

all graphs is hard [72]. We notice similar behavior with Gemini. Our framework does not employ any partitioning algorithm for load balancing and still performs well in practice.

3.6.9.2. *Galois.* Galois [91] is shared-memory graph analytics platform that implements amorphous data-parallelism. Here graph operations are performed on *active nodes* by executing an *operator*. Each active node can access certain graph elements that consists of its *neighborhood*. Operators can be implemented in either pull or push style. Vertices are activated by two different scheduling techniques: autonomous and coordinated scheduling. Tasks in Galois can be scheduled based on the machine topology. In addition, autonomous scheduling can assign priorities to tasks so as to decide which one to execute first.

We report the best shared-memory performance of Galois for SSSP problem in Figs. 3.32 and 3.34 with Friendster and sk2005 inputs. Such comparison helps us to understand the overhead associated with communication and scheduling in a distributed-memory graph system. We also conducted experiments with connected component algorithms in Galois. Unfortunately, the verifier fails with all graph inputs in Galois in this case and we have decided not to include these results.

3.7. Related work

To utilize abundance of parallelism in runtimes, frameworks based on Think-Like-A-Vertex (TLAV) has to make several choices in terms of the following metrics [84]:

a) Temporal scheduling aspect: Traditional Bulk-Synchronous-Parallel (BSP) model for graph computation such as predominant Gather-Apply-Scatter (GAS) approach ([55, 82, 76]) divides an algorithm execution into several supersteps. Within each superstep, computation and communication for the current superstep can progress independently. However, the updates from the previous superstep are applied to the next superstep after a global synchronization barrier and workers can not progress to the next superstep until all the computations and communications for the current superstep has finished. *Synchronous* algorithms with BSP model suffer from the straggler effect and also fail to converge in some cases (for example, graph coloring [111]).

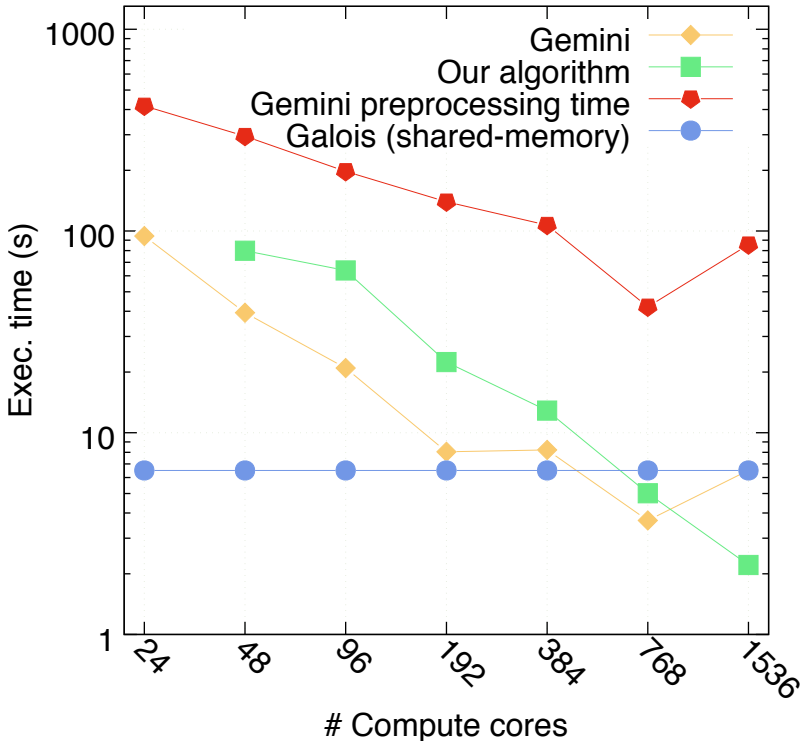


FIGURE 3.32. Comparison of SSSP algorithms with Friendster dataset.

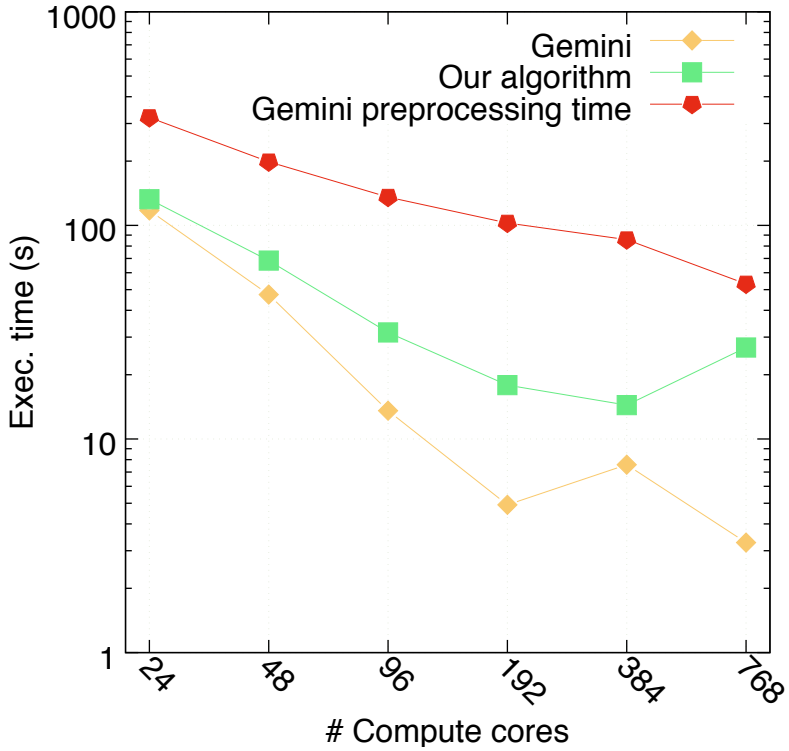


FIGURE 3.33. Comparison of SSSP algorithms with Twitter dataset.

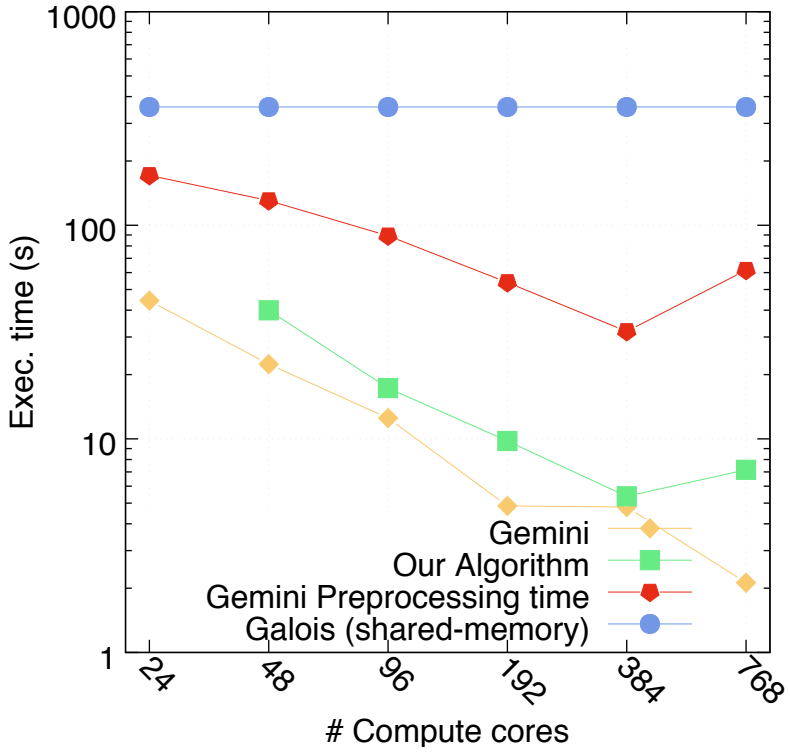


FIGURE 3.34. Comparison of SSSP algorithms with sk2005 dataset.

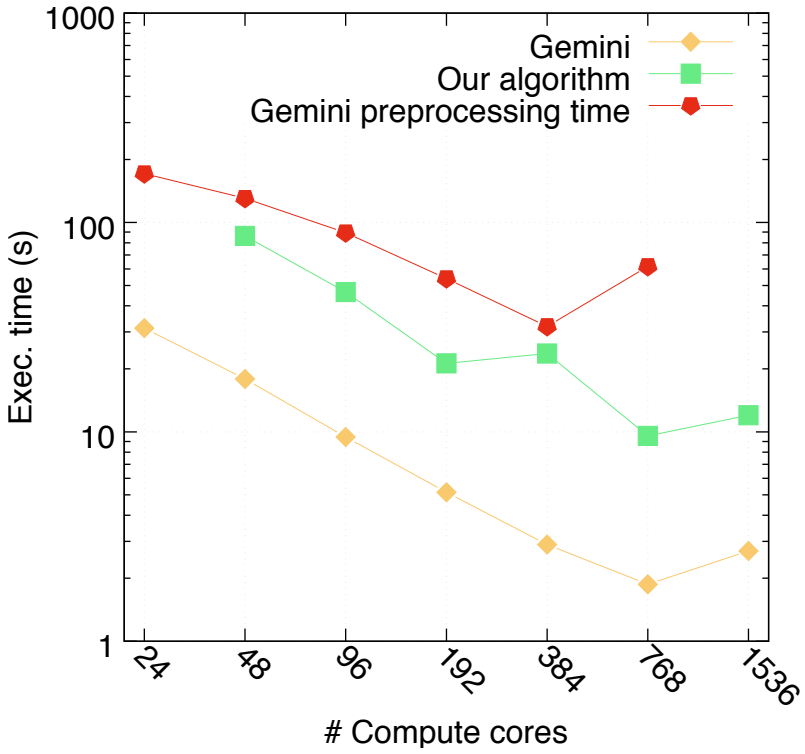


FIGURE 3.35. Comparison of connected component algorithms with sk2005 dataset.

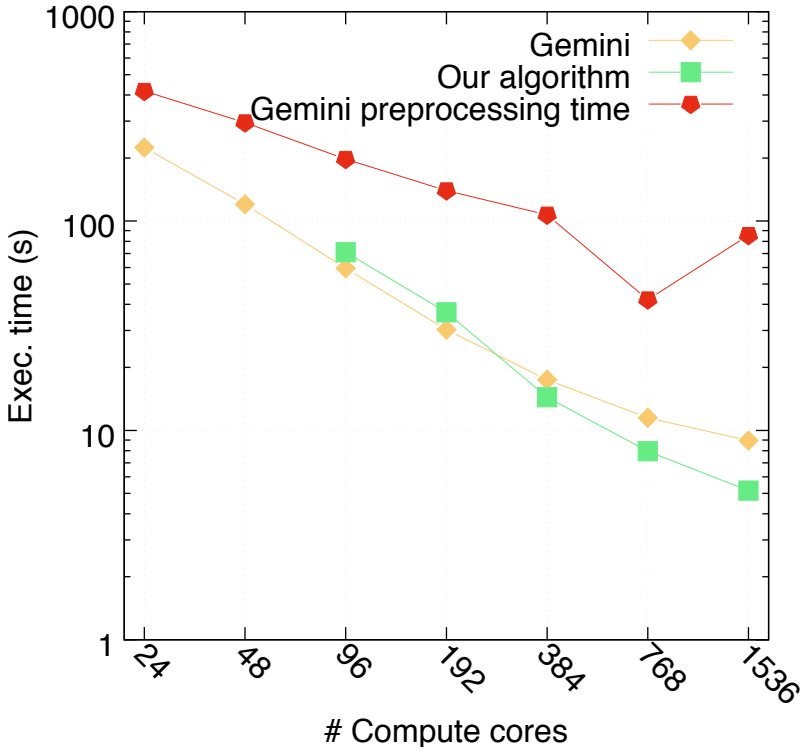


FIGURE 3.36. Comparison of connected component algorithms with Friendster dataset.

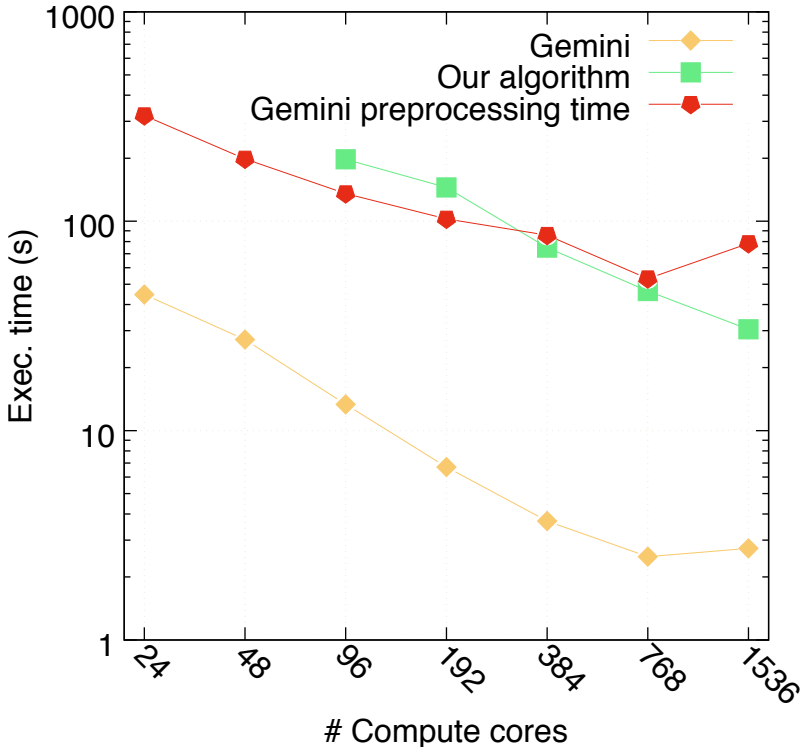


FIGURE 3.37. Comparison of connected component algorithms with Twitter dataset.

To address the shortcomings of GAS approach, TLAV framework PowerGraph provides an *asynchronous* mode of graph execution engine. In contrast to the synchronous engine, which requires barrier after each of the GAS micro steps, the asynchronous engine can start computation on an active vertex as soon as the updates are available. However, such asynchronous execution in GAS model requires the runtime to acquire lock on an active vertex before executing the GAS steps. Doing so ensures that no neighboring vertex is active at the same time. With power-law graphs that contain high-degree vertices, such way of asynchronous execution can quickly become a bottleneck in distributed setting [84, 111]. Instead of complete asynchrony, some stricter variations of asynchrony [58] have been proposed: for example, k-level asynchronous [60] (asynchrony based on graph "topology") i.e. how many vertices on a path (levels) can be explored in advance before synchronization) and Δ-stepping [85]. (vertices with tentative "property" values such as distances of vertices from the root in the range $[c\Delta, (c+1)\Delta]$ that can be explored in parallel before synchronization). Additionally, some hybrid execution frameworks have been proposed [111]. However these restricted and hybrid asynchronous execution models rely on preprocessing of input and pre-execution of graph algorithms on a particular graph input either to learn the parameters $(k, \Delta$ etc.$)$ or to train a prediction model with machine learning.

Maiter [114] proposed a delta-based accumulative asynchronous iterative computation (DIAC) that is only shown to be limited to applications with *monotonic* updates. In contrast, our approach is more general that can be applicable to a broader class of graph algorithms including non-monotonic updates. In addition, the termination detection algorithm in Maiter is based on an inefficient master-worker synchronous termination check, where each worker periodically has to send update report to the master. This defeats the purpose of truly asynchronous execution and can become a bottleneck for scalability.

Stale Synchronous parallel (SSP) model [64], a relaxed version of BSP approach, allows computation to proceed locally based on stale cached values within a specific bound (termed as *bounded staleness*). However, SSP performs significant stale computation and

it is challenging to find suitable cache update frequency to reduce stale computation. A recent approach [47] proposed to accumulate changes and decide on-the-fly when to start next synchronous iteration. This requires messages to be buffered for next iteration and can be problematic for power-law graphs with large number of neighbors. None of [114, 64, 47] report scalability beyond 500 cores.

Another framework related to our work is due to [96, 94, 95]. Their work is based on specialized graph processing framework HavoqGT (Highly Asynchronous Visitor Queue Graph Toolkit) which is particularly tailored for graph processing. This framework supports graph-specific load balancing, routing and management for algorithmic events. In particular, HavoqGT spends significant amount of pre-processing time for partitioning the input graph to handle load imbalance. In contrast, our framework does not re-distribute the graph for load balancing purpose, hence spends little time in pre-processing the graph (to remove self-loops for example). Instead, we rely on algorithm design techniques and runtime support to achieve load balancing on-the-fly. Another difference between HavoqGT framework and ours is that the visitor queues are centralized while we employ thread-local priority queues. HavoqGT also supports large graph processing using mmap as external memory. In comparison to their work, our work focuses on the performance of graph algorithms in general-purpose AMT runtimes with graphs that fit in main memory. Although SSSP and CC algorithms have been evaluated in [94] for shared memory systems, no evaluation is available in distributed settings for connected component and no comparison has been done for SSSP with most well-known Δ-stepping algorithm. Additionally, implementation and evaluation of these algorithms on a general purpose asynchronous many-task (AMT) runtime have not been evaluated. As will be discussed in the thesis, in our experience, graph algorithms need significant runtime support to achieve better performance on AMT runtimes.

b) Communication framework: Depending on how data is shared across the system, communication frameworks can be divided into three categories. In *message-passing* frameworks such as Pregel, processes communicate messages synchronously to ensure

data consistency. To avoid the constraints associated with synchronous message-passing system, Graphlab and the next generation of Powergraph are implemented as *distributed shared-memory* TLAVs. However, due to the shared-view of the data across different processes, asynchronous execution in such framework mandates locking active vertices that can limit scalability [111]. This model can suffer performance bottleneck due to the requirements such as memory consistency to avoid false-sharing, and over-contention of the network interconnect. A promising alternative is *active messages* [105], where messages are sent explicitly but receives are implicit. Here, since a pre-registered receive handler knows a priori the address of user-level handler, it can directly extract the computation from a message and integrate it to the ongoing computation. In doing so, active messages (AM) eliminate software overhead of buffering and allow for more overlapping between communication and computation and expose more asynchrony.

c) Program execution model: Closely related to temporal scheduling and communication is the program execution model. Vertex-centric programs can be executed iteratively in combination of one or more of the three main phases: Gather (accessing the data), Apply (performing computation), Scatter (propagating the update). This execution model is considered to be the "pull" model of computation, since the active vertices pull data from the neighboring vertices before computation. Asynchronous execution in such model cannot unveil the advantage of sender-side combiner [84].

3.8. Conclusion

In this chapter, we demonstrate the benefit of unordered distributed graph algorithms, implemented with active messages, that exploits optimistic parallelization by eliminating global and vertex-centric barriers and by removing ordering constraint. The time gained by the exclusion of global and vertex-centric synchronization can be invested in local ordering and can result in better performance with graphs that have high-degree vertices and densely connected sub-communities.

CHAPTER 4

Context Matters: Distributed Graph Algorithms and Runtime Systems

4.1. Introduction

Large, irregular applications are gaining recognition as the future challenge in parallel computing. This is reflected by the Graph500 benchmark [88], the subject of which is the prototypical irregular problem of graph traversal. Graph traversal is a basic building block of other graph algorithms used in social network analytics, transportation optimization, artificial intelligence, power grids, and, in general, any problem where data consists of entities that connect and interact in irregular ways. The current Graph500 benchmark is based on breadth-first search (BFS) with a proposal to extend the benchmark with single-source shortest paths (SSSP). In this work, we concentrate on BFS and SSSP for the same reasons, i.e., as representatives of a class of irregular graph problems.

Research on distributed graph algorithms (DGA) is an emerging and active field. New algorithms, new approaches to distribute the data, and new performance results appear at most major distributed computing conferences. The Graph500 benchmark bears witness to the progress with the best results progressing from seven GTEPS (billions of traversed edges per second) in 2010 to 23 TTEPS in 2014. Many new algorithmic techniques have been developed, e.g., direction optimization [16, 15], pruning [27], k-level asynchronous algorithm [61], hybrid algorithms [27], and distributed control [113]. A practitioner faces a multitude of published approaches, which are often vague on low-level details of implementations.

However, for DGAs, the low level details of implementation details profoundly affect performance. This is because DGAs exhibit little locality, rarely require any significant computation per memory access, and result in a high-rate of communication of small messages. Thus, unlike regular algorithms that are built on top of well-understood regular

communication and memory access, graph algorithms interact with the entire software and hardware stack in a complex way due to their data-driven, fine-grained, irregular nature of tasks. Each piece of the stack, designed independently, from the algorithm level through the transport layer to the hardware layer and the topology of the physical network, interacts within the system. This makes designing DGAs an experimental endeavor, and this state of affairs will be only exacerbated as we move towards exascale computing. It should be noted that the complexity of the interactions between high-level algorithm and low-level runtime is not unknown to the practitioners. However, this knowledge is implicit, fragmented, and often sidelined in presentation of new techniques. Notably, authors in [30], who achieve the top results in the Graph 500 benchmark in part due to direct access to the SPI (System Programming Interface) low-level primitives, provide an outstanding analysis of their evolving implementation, including a three-years timeline of changing conclusions and understanding. Unfortunately, this manner of reporting is not typical for the field.

To draw attention to the importance of runtime concerns, we argue that the topic of interplay of runtime and algorithmic concerns is too large for any single researcher to tackle; it is necessary that the community develops knowledge collectively. To this end, a common ground is needed.

TABLE 1. Runtime-Level aspects of DGAs. These aspects need to be disclosed when reporting results; for those that are quantifiable, numerical values should be reported for each experiment.

Transport		Sec. 4.2.1.1
Communication Paradigm	Point-to-Point[30, 113, 7], Collectives[24, 23, 30, 93]	
	One-Sided[7], Active messages[113, 27]	
Request Tracking	Remotely synchronous, Asynchronous[113, 69]	
	Locally synchronous[69, 30]	
Progression[30, 113]:		

• Asynchronous	System threads[113], User threads
• Synchronous	Explicit progress[113], Runtime scheduler[113]
	Lightweight task[113]
Bit Transport	System Processing Interface (SPI)[31, 27, 30]
	Message Passing Interface (MPI)[23, 30, 24]
	Active Pebbles[109], ARMI[59, 61], Photon[69]
Protocol	Eager, rendezvous[113], completion[27, 30]
Optimization	Message reduction/caching[96, 113, 93]
	Message coalescing[30, 96, 95]
	Message compression[30]
Message routing	2D, 3D[95, 96], ring[112], hypercube, rook[42]
Thread safety	Multi-threaded[113], serialized, funneled

Network Topology Sec. 4.2.1.2

Physical Topology	3D[95, 29] and 5D[29] torus, Dragonfly[113, 24]
Logical Topology	Skewness[24], synthetic network[95, 96]
Job topology	Job allocation, rank mapping[24]

Local Scheduling Sec. 4.2.1.3

Threading	Pthreads[113], OpenMP[23, 24], HPX-5 [7]
Task management	Work stealing, FIFO/LIFO tasks[113]
	Priority scheduling[7], System threads[29]
Termination	Quiescence detection[95], SKR[113]
Heuristics	End-epoch frequency[113]
	Eager progress limit[113]
Hardware effects	L2 atomics[27], NUMA effects[24]

Runtime Feedback

Optimal K-level[61], optimal Δ, sync to async switching[111]

We contend that the advancements in the field are difficult to generalize and reconcile because the information reported is commonly incomplete. The low-level details of implementations are often vague or missing. Yet, these can have important impact.

In this work we propose a template of runtime features, presented in Table 1, to aid authors in reporting their work. We arrive to this template by combining our own experience in co-designing runtime system and DGAs with findings from review of existing literature. For completeness, we also list algorithm-level aspects, Table 2. We invite others to extend and revise our suggestions, and to make this a community effort. Widespread adoption of our recommendation will enable transferability of lessons learned across the field, metastudies of the interaction of runtime and algorithmic concerns to potentially derive abstract models, and, with deepened, systematic understanding, improvement of performance.

4.2. Analysis of DGAs

In this section, we analyze and describe a sample of the existing research on distributed BFS and SSSP problems. Our motivation is two-fold. First, we want to get an overall feel for how complete (or incomplete) is the information about runtime part of the DGA stack that is presented in literature. Second, we aim to identify potential aspects of DGAs beyond those stemming from our own work. DGAs consist of complementary *runtime-level* (Sec. 4.2.1) and *algorithm-level* (Sec. 4.2.2) aspects. Algorithm-level aspects vary for different graph algorithms; however, the runtime-level aspects are algorithm agnostic. For this reason, our analysis of runtime concerns is applicable to any DGAs. The purpose of our effort is to construct a blueprint for a more holistic treatment of DGAs. Tables 1 and 2 summarize the runtime-level and algorithm-level aspects of DGAs discussed in the remainder of the section. Moreover, Table 1 serves as a template for reporting runtime aspects of DGAs, and we use it as such to describe three DGA runtimes in Table 3.

TABLE 2. Algorithm-Level aspects of DGAs. These aspects are usually disclosed in literature; we present them here for completeness.

Approach	
Ordered	Level-Synchronous[112, 30]
	Bellman-Ford[93]
	Combinatorial BLAS[23, 16, 24]
Ordered/Unordered	Hybrid[27], HSync[111]
Unordered	Distributed control[113]
	KLA[61], Δ-stepping [43, 93]
Algorithmic Considerations	
Data Distribution	1D[43, 93, 27, 113, 30]
	2D[112, 23, 24], Edge list[95, 96]
Optimizations	Ghosts[55, 76, 95, 96], Pruning[27]
	Direction optimization[27, 16, 30, 24]
	Priority messages[113]
	Tree-based broadcast, reduction, and filtering[96]
Load Balancing	Per-thread work splitting[27],
	Random shuffling of vertex identifiers[23]
	Delegates[96], Proxies[27]
Graph Representation	
CSR[23, 16, 93, 113, 24], compressed coarse-index adjacency list[30], skip list[30], doubly-compressed sparse column (DCSC)[23, 24]	
Data Structures (algo. progress)	
Distributed async visitor queue[95, 96], thread-local priority queue[94, 113], dynamic-array buckets[93]	

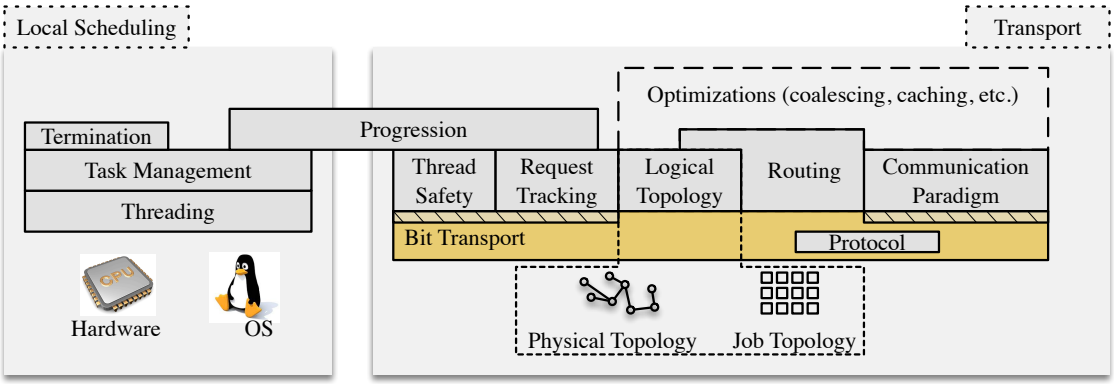


FIGURE 4.1. Overview of the runtime stack components

4.2.1. Runtime-Level aspects of DGAs. DGAs *runtime* (Fig. 4.1) has two major distinguishable components working in rapport: a *transport* (Sec. 4.2.1.1) and a *scheduler* (Sec. 4.2.1.3). The basic component of a transport is a *bit transport* that is a system-provided implementation of an interface for actually sending bits over the wire. Bit transport can support different *protocols* for exchanging messages such as eager or rendezvous protocols. The bit transport API may support levels of *thread safety*, impacting both the use and the internal working of the transport (hence the shared part in the figure). *Request tracking* is the mechanism for making and keeping track of communication requests. Building on bit transport, the runtime must use the request tracking mechanism in some way, according to thread safety, to initiate and complete requests. We call this process *progression*. The choice of *Communication paradigm* has reverberating impact both on the rest of the transport layer and the DGA itself. For example, collectives require more memory for communication results than point-to-point communication and the progression mechanism has to be designed with fewer but more heavyweight requests in mind. Transport may provide *logical topologies* that provide additional *routing* on top of *physical* and *job layout* topologies to improve performance (e.g., message reductions in logical topology) or to reduce memory requirements (e.g., fewer coalescing buffers). Finally, the transport may employ various optimizations such as message coalescing.

The *scheduler* part of the runtime schedules worker threads or tasks. One of the responsibilities of the local scheduler is to ensure transport progression. Local scheduler can also be responsible for checking termination of an algorithm.

In the following discussion, we categorize important runtime-level parameters based on the review of existing literature for DGAs. Table 1 summarizes the set of low-level runtime parameters.

4.2.1.1. *Transport.* The transport layer is the part of the stack responsible for sending and receiving bits. Important properties of transport include how message buffers are handled, which entity manages them, and how frequently they need to be managed. The runtime needs to take several decisions regarding these.

The choice of ***communication paradigm*** can have a notable impact on performance of DGAs. Each paradigm imposes different trade-offs in terms of memory constraints, synchronization overhead, and network latency. The *collectives* paradigm is used when large, low-overhead stages of all-to-all communication are needed, *point-to-point* paradigm allows for finer overlap between computation and communication at the expense of code complexity, and *active messages* are a refinement of point-point communication that adds an implicit execution of *handlers* on remote objects. Finally, *one-sided* paradigm provides remote memory operations (GET, PUT, etc.) which are very efficient but require remote memory management protocol.

For example, collectives are the base of BLAS approaches and level-synchronous approaches. However, [30] show how using lightweight point-to-point communication may lead to improvements in traditionally synchronous approaches. They compare their point-to-point implementation using low-level BlueGene/Q's System Processing Interface (SPI) to an MPI implementation using collectives. They note the large memory footprint required for collective buffers, which forces them to decrease the scale of the problem per node. Furthermore, collectives do not allow for easy interleaving of computation with communication. Active messages are based on point-to-point communication, and they display similar communication performance characteristics.

Request Tracking (RT) refers to how communication requests are made: a request is scheduled and completed separately (*asynchronous*), a request is made and completed at the same time (*locally synchronous*), or a request is made and the requester waits until it has been

completely processed (*remotely synchronous*). As an example, *remotely synchronous* request tracking in MPI (e.g., by `MPI_Ssend` etc.) guarantees a small number of messages on the network, but it hinders parallelization. On the other hand, using *locally synchronous* RT (e.g., `MPI_Send`) makes it easy to reuse buffers and may allow parallelism if eager protocol is used. Finally, *asynchronous* RT uses interfaces such as `MPI_Isend/MPI_IRecv` to start requests along with interfaces such as `MPI_Testsome` to check for their completion, maximizing overlap (if the MPI implementation supports it) between computation and communication at the cost of more complex progression and request management.

Completing a round trip through transport requires bookkeeping, performing bit moving, and delivering the results of completed requests—we call that ***progression***. Progression influences the timeliness and efficiency of transport delivery, and a wrong progression model can render an algorithm infeasible (Sec. 4.4.3). *Asynchronous* progression is performed periodically and is scheduled through dedicated resources such as *system* or *user* threads. For example, Cray MPI provides an option for starting progression pthreads that perform internal MPI progress in parallel with the algorithm threads. Progression through *user threads* is scheduled by the runtime explicitly, and, for example, calls MPI repeatedly to generate progress. In contrast, *synchronous* progression is initiated periodically from the runtime. In *explicit* progress, the algorithm can choose, bypassing the runtime scheduler (if any), when to call progress, enabling optimizations at the cost of added code complexity. For example, in [113], we employed explicit polling in our Distributed Control (DC) algorithm for SSSP, but observed a decrease in performance. In a task-based system, network progress can be scheduled as a *lightweight* task. For example, AM++ [109] implements network polling, buffer flushing, checking for termination, and executing pending handlers for received messages as tasks, on equal footing with algorithm tasks that run message handlers. HPX-5 [7] executed network progress in a similar fashion, but the more recent versions switched to explicitly initiating progress in the main *scheduler loop*, giving the runtime more control over when progression is executed. Most authors

do not discuss progression and request tracking explicitly, but the choices made for these parameters may have a profound effect on performance (cf., Sec. 4.4.3).

Bit transport is the lowest-level network interface used by upper levels to deliver bits from one location to another. In efficient BlueGene/Q implementations [31, 27, 30], the *System Processing Interface (SPI)* communication layer serves as a bit transport (as described above). The majority of implementations in the literature use *Message passing Interface* (MPI) for their bit transport. SPI is a direct interface to hardware queues, while MPI is a complex framework with extra functionality and semantics. Direct interfaces such as SPI may yield more efficient communication, but are less or not at all portable, and may require more implementation effort. The third type of bit transport is based on remote method invocation (RMI) technique and is used in approaches based on STAPL [59, 61], a generic parallel library for graph and other data structures and algorithms. STAPL uses the ARMI (Adaptive Remote Method Invocation) active-message communication library, based on RMI. ARMI supports automatic message coalescing but does not provide routing or message reductions natively. HPX-5 runtime has support for two types of bit transport: one is based on MPI, another one is based on Photon [69] RDMA middleware library. Photon is based on RDMA put and get *with completion*, where requests are completed asynchronously, and their completions are written to *ledgers* that can be read by higher level runtimes. HPX-5, AM++ and ARMI can use different bit transport backends, making the interface boundary between the bit transport and the runtime very clear.

Bit transport may employ different ***protocols***. For example, MPI point-to-point communication may support *eager* protocols for small messages and *rendezvous* protocols for larger transfers, sending messages without or with, respectively, round-trip communication [9]. The choice of protocols may have a detrimental impact on algorithms (e.g., Sec. 4.4.2), and it may be difficult to control explicitly. For example, most MPI implementations provide extensive configuration options, but these options are not standardized and often can only be fully utilized by experts.

A number of runtime-level ***optimization*** techniques have been proposed in the literature to reduce communication overhead and maximize throughput. AM++ provides *message reduction* (caching). [96] used tree-based broadcast, reduction and filtering for communication involving high degree vertices. [93] used local lookup arrays to track the tentative distance of every vertex, thus avoiding duplicate request being sent.

Increasing *message coalescing* (cf., Sec. 4.4.2) buffer size increases the rate at which small messages can be sent over a network at the cost of latency. [30] use coalescing to pack together all the edges that would be sent to each destination separately and queue them in an intermediate buffer. [96, 95] combined coalescing with routing to reduce dense communication.

Message routing constructs a logical topology to add intermediate targets for messages. [95] implemented routing through a synthetic network to mimic the BG/P 3D torus interconnect topology. In a follow-up paper [96], the authors additionally embedded the delegate tree as a means for further communication reduction. AM++ [108] supports software routing and provides two predefined strategies: *rook routing* and *Hypercube routing*. Rook routing reduces the number of communicating buffers to $O(\sqrt{p})$ [42]. [112] used ring communication in their optimized collective implementation and adjusted the diameter of the ring to achieve better performance.

Thread Safety A message passing framework can support different levels of thread safety. For MPI, there are 4 levels in total [10]: *single*, *funneled*, *serialized* and *multiple*. We used multiple as the threading level together with an asynchronous progress thread in our AM++ *DC* [113] implementation.

4.2.1.2. *Network Topology.* Computing resources are organized in several specific *physical topologies*: 3D torus, dragonfly, 5D torus, and so on. The physical topology impacts the efficiency of communication in a graph computation. For example, Cray MPI provides an all-to-all implementation that is optimized for Aries and Gemini systems. In another example, [29] map parts of graph adjacency matrices onto the Blue Gene/Q 5D torus

topology in such a way that neighboring parts of the matrix are also neighbors in the physical topology.

The *logical topology* is the layout of the data in physical topology. [24], for example, found that *processor grid skewness*, i.e., the distribution and the shape of the blocks of an adjacency matrix, had significant impact on their results: the "tall skinny" grids (more blocks across the Y dimension of the matrix) performed faster, and "short fat" grids (more blocks across the X dimension of the matrix) performed worse than square grids.

Job scheduler for computing resources allocates nodes based on scheduling policy resulting in a *job topology*. [17] showed that job topology may have an impact on performance due to the distances among allocated nodes or due to contention on shared network.

4.2.1.3. *Local Scheduling.* Depending on the node-level threading mechanism, thread scheduling policies, and synchronization primitives, tasks associated with a DGA can execute in different order with varying frequencies. For example, in an attempt to quickly spread good work, a message can be sent with priority and put the message handler in front of the task queue [113]. Supporting data structures, for example bitmaps in sync mode and global queue in async mode in [111], can be another way to implement local scheduling. Below, we discuss several thread-granularity and scheduling-related factors.

Threads (worker threads) can be used for intra-node threading. [23] and [24] used MPI for inter node processing and GNU OpenMP for intra-node threading. [113] used a combination of MPI and pthreads. HPX-5 uses suspendable lightweight threads with their own stacks and with cheap thread transfer.

Lightweight threads or tasks, implemented on top of kernel threads, can be scheduled differently. *Task management* mechanisms achieve load balancing by mechanisms such as *work stealing* and *FIFO/LIFO* schedulers. Our HPX-5 implementation also provides a *priority* scheduler for algorithms that can exploit it.

4.2.2. Algorithm-Level aspects of DGAs. Table 2 summarizes the set of parameters we identified as the algorithm-level aspects of DGAs, and divide them into four categories. The *approach* category is about the main algorithmic choices, the *algorithmic considerations*

category covers the main aspects of the approach, and the categories of *graph representation* and *data structures* cover the data structures that are used.

4.3. Our template in practice

In this section we show how our proposed template can be applied in practice by comparing 3 different runtimes: AM++, HPX-5, and the IBM BlueGene/Q implementation. The characteristics of the runtimes that we consider are independent of any particular application. Our recommendation is to consider DGAs holistically, encompassing runtime and algorithmic concerns. Specifically, we apply the template to Distributed Control (*DC*) algorithm[113] for solving the SSSP problem implemented in the AM++ and HPX-5 runtimes and to BlueGene/Q BFS [30]. The BlueGene/Q BFS is a rare case where the authors disclosed enough information so that we can fill out a "report card" based on our template (Table 3). We choose *DC* because it is particularly well suited as a subject of inquiry into interplay of runtime and algorithmic concerns, as will be evident from next section devoted to experimental results.

TABLE 3. Properties of the AM++ [113], HPX-5 [7], and BlueGene runtimes [30] summarized in a template based on Table 1.

HPX [7]	**AM++** [113]	**BlueGene/Q BFS** [30]
	Transport	
Paradigm		
Parcels	AM++ [108] active messages.	One-sided (low-level active messages).
Request Tracking		
With Photon, sends are asynchronous, and receives are automatic with completion events. With MPI, asynchronous MPI interfaces are used, with limited number of active send and receive requests. Send requests are queued until an MPI send slot becomes available. **Required**: Send/receive limits (MPI), ledger size (Photon)	Sends and receives are scheduled with asynchronous MPI interfaces. The number of receive requests is kept constant, and send requests are created on demand with a flow-control mechanism to cap the number of outstanding requests. **Required**: number of receive buffers, flow control limit	Sends are locally synchronous (through SPI) with one buffer per destination (maximum one outstanding send per destination). Reception is asynchronous through polling of SPI counters.
Progression		

HPX-5 invokes progress explicitly in the scheduler loop. Progress is invoked by a worker thread if there is no local work left, and with the priority scheduler for DC, progression is invoked periodically based on flow control feedback. Progression for MPI is serialized between workers. Sends are processed first from a send queue, then receives are processed and reused as they complete. For Photon, progression can be run by multiple threads (Photon is thread safe). Completion events returned from Photon are processed. **Required**: Flow control parameters	AM++ uses shared coalescing buffers. Requests can be explicitly polled (with `MPI_Testsome` on an array of requests). AM++ also has special purpose user-level tasks that perform progression, executed from AM++ task queue when sends are performed, or when end-of-epoch tests are performed (during these tests AM++ tries to finish an epoch by processing remaining work). On Cray machines, distributed control performs the best when an asynchronous MPI progress thread is used. **Required**: number of poll tasks	A lightweight asynchronous communication layer on top of System Processing Interface (SPI) with separate FIFOs for injections and receptions (multiple queues per node, providing network-level parallelism). Messages are coalesced to per destination buffers where each buffer is exclusively owned and operated on by a single thread, eliminating locking and contention. Buffers are placed into injection queues when ready, and a thread will wait for completion when another message needs to be sent to a given destination. **Required**: number of FIFO queues
Bit Transport		
MPI (high-level), Photon (low-level).	MPI, a high-level portable interface.	SPI, a low-level BG/Q-specific interface.
Protocol		

Depends on coalescing sizes. MPI: depends on implementation. Photon: direct and rendezvous.	Depends on MPI implementation and coalescing buffer sizes (Sec. 4.4.2). **Required**: list of used protocols	FIFO injection queue send and receive in SPI.
Optimization		
Message coalescing (per destination or cumulative) and compression. **Required**: Coalescing buffer sizes	Message coalescing. AM++ is capable of reduction and caching, but these optimizations are not beneficial for DC. **Required**: Coalescing buffer sizes	Message coalescing and compression. Compression is beneficial only for large BFS wave fronts. **Required**: Coalescing buffer sizes
Routing		
No routing.	Rook, hypercube routing.	No routing.
Thread Safety		
MPI: thread serialized. Photon: multiple threads.	AM++ works with MPI thread serialized and thread multiple safety levels.	Multiple threads can progress SPI FIFO queues.
Network Topology		
Physical Topology		
Star (central switch).	3D torus Gemini and Dragonfly Aries.	BlueGene/Q 5D torus.

Logical Topology		
Global Address Space (PGAS or AGAS).	None.	None.
Job Topology		
Inconsequential because of star topology.	Unknown (execution time averaged from the same batch execution).	Unknown. Largest runs show significant variability.
Local Scheduling		
Threading		
Pthreads, lightweight user threads.	Pthreads.	Heavyweight worker threads.
Task Management		
LIFO and priority queues of *parcels*, which represent undone work or suspended threads.	FIFO queue with every coalesced buffer represented as a task.	None, no lightweight tasks.
Termination		
SKR termination detection (activity counts periodically reduced).	SKR termination detection (activity counts periodically reduced).	Unknown. "Termination check" is mentioned at least once.

4.3.1. The Runtime Report Card. The "report card" in Table 3 enumerates the runtime features of each of the two DGAs according to the template in Table 1. We summarize the runtimes in a table form for clarity, but we do not advocate that scientific publications use that exact format. The important task is to ensure that all relevant aspects of the runtime are adequately covered. Furthermore, every runtime feature must list relevant associated

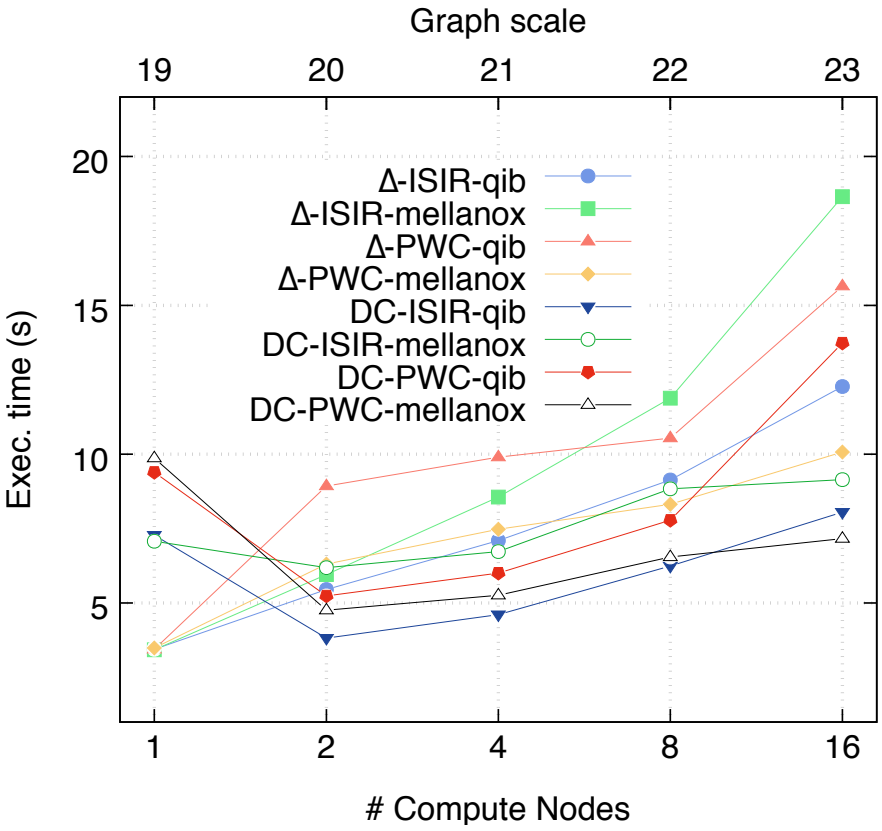


FIGURE 4.2. Execution time of Δ-stepping and DC on HPX-5 with two different bit transports (ISIR and PWC) and two different interconnects: InfiniPath QLE7340 (qib) and Mellanox ConnectX-3 EN (mellanox).

quantities that are necessary for complete interpretability of experimental results. For example, in presence of coalescing, performance results cannot be interpreted if the size of coalescing buffers is not given.

4.4. Runtime Parameters of DGA Performance

DC is particularly sensitive to the runtime characteristics such as timing of task execution, communication latency, and buffering, and we use it to illustrate the impact that the runtime may have on the performance of DGAs. In the remainder of this section, we discuss the characteristics of the runtime used in our experiments and the impacts we observed. Our experiments were run at varying times on different machines. The experiments we discuss here were ran on Big Red 2 (BR2) at Indiana University [1], on

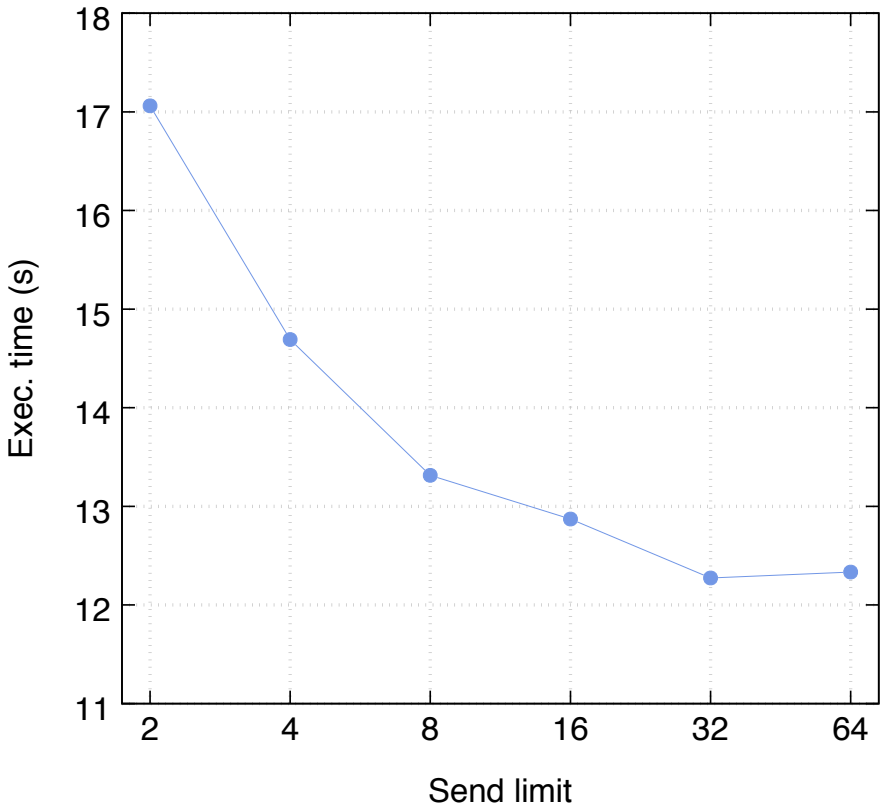


FIGURE 4.3. Impact of send limit for ISIR network on the InfiniPath interconnect with 16 nodes and scale 23 graph for Δ-stepping algorithm.

Edison at NERSC, and on our own cluster Cutter consisting of 16 nodes with 16 Haswell cores each and equipped with InfiniPath and Mellanox interconnects. The most important differences between the two Cray machines are the topologies – 3-D torus on Big Red 2 vs. Dragonfly on Edison – and the MPI implementations – Cray's Message Passing Toolkit (MPT) 6.2.2 on Big Red 2 vs. MPT 7.1.1 on Edison. On Cutter, we use OpenMPI 1.8.8 for MPI runs and Photon [69] otherwise. On Cutter, we also vary the interconnect between InfiniPath and Mellanox. All experiments were run on Graph500 graphs. All execution times are reported in by taking the average of executing multiple problem instances (over the same batch job execution).

4.4.1. Effect of Bit Transport and Interconnect. Figure 4.2 demonstrates how different bit transports (MPI ISend-IRecv based ISIR transport and Photon Put-With-Completion (PWC) [69] transport) can affect the performance of an algorithm. We have ran Δ-stepping

n	qib	mel
1	4	8
2	8	32
4	8	128
8	16	256
16	32	64

TABLE 4. Optimal send limits for weak scaling experiment in Fig. 4.2 for InfiniPath (qib) and Mellanox (mel) interconnects.

and DC with HPX-5 on cutter for these experiments. In terms of performance and scaling, DC with PWC transport and Mellanox interconnect performs the best. In fact DC shows better scaling for 8 and 16 nodes with Mellanox than with InfiniPath for both PWC and ISIR transports. Our one node performance is taken with networking turned on; DC performs much worse than Δ-stepping, but it quickly improves with scale. Δ-Stepping does not show good scaling behavior altogether.

While experimenting with ISIR transport, we have tried different limits for the number of MPI `Isend` calls that HPX-5 spawns concurrently. Figure 4.3 shows how varying the send limit changes the performance of Δ-Stepping algorithm for one of the scales. Table 4 shows the optimal send-limits for Δ-stepping algorithm with ISIR transport. This experiment illustrates how a runtime-level bit transport parameter can make an impact on the performance of an algorithm.

4.4.2. Effect Of Coalescing Size On Transport Protocol. To increase bandwidth utilization, AM++ performs *message coalescing*, combining multiple messages sent to the same destination into a single, larger message. Messages are appended to per-destination buffers. To handle partially filled buffers, a periodic check is performed to check for activity. In the case of *DC* SSSP, a single message consists of a tuple of a destination vertex and distance, 12

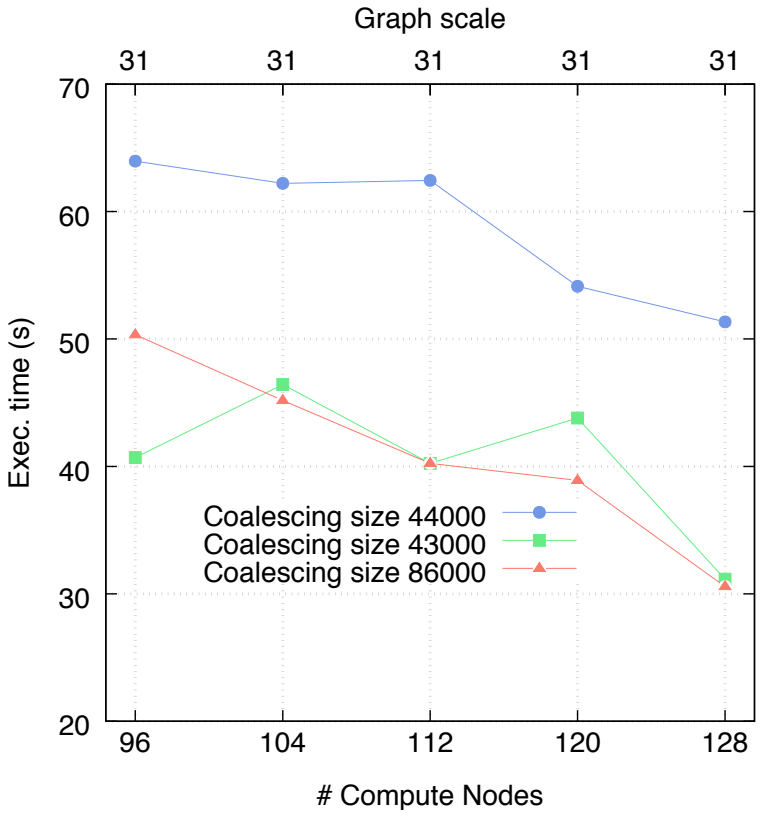


FIGURE 4.4. Effect of coalescing on DC SSSP algorithm with scale 31 graph (BR2).

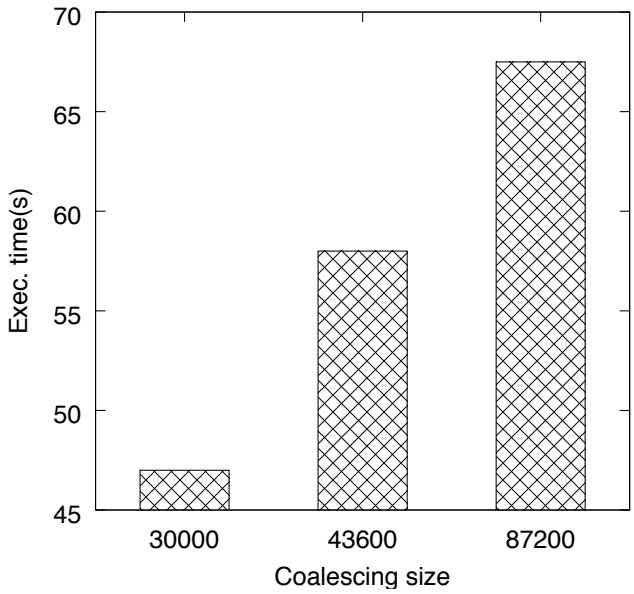


FIGURE 4.5. *DC* BFS algorithm.

FIGURE 4.6. Effect of coalescing on DC BFS algorithm with scale 31 graph (BR2).

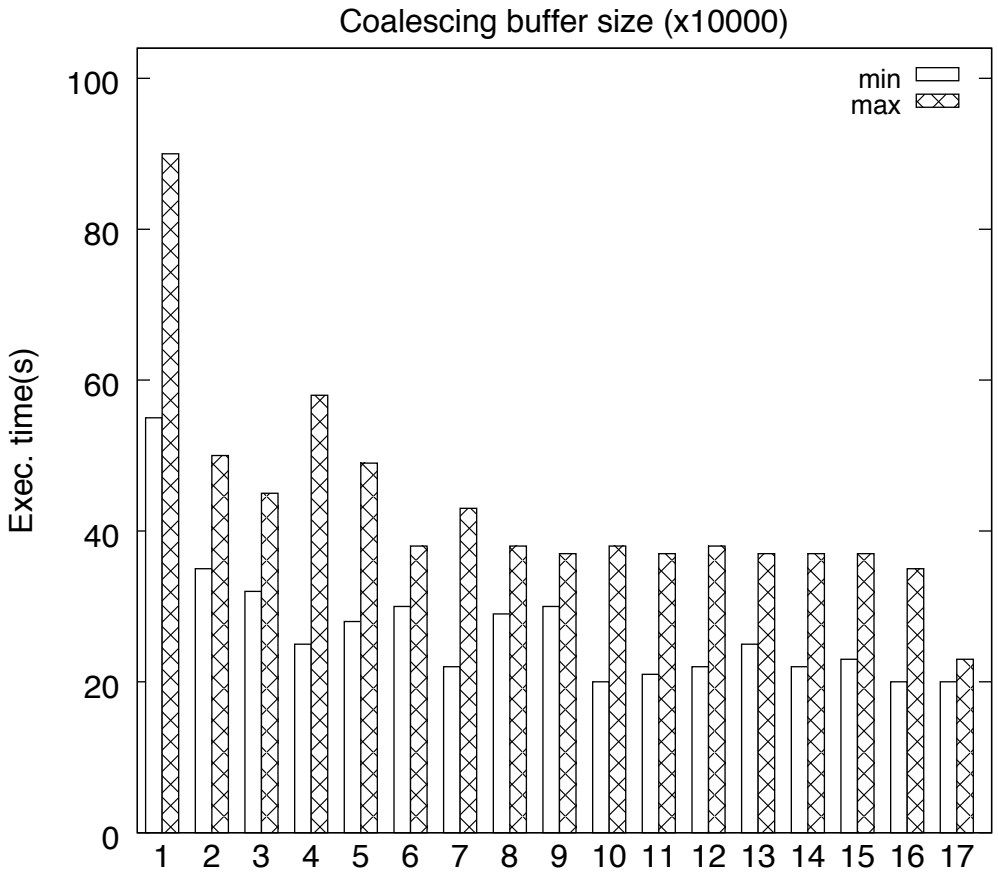


FIGURE 4.7. Effect of coalescing size on *DC* SSSP algorithm on a scale 31 graph (Edison).

bytes in total. With such small messages, coalescing has great impact on the performance, but finding the optimal size is difficult.

We investigated the impact of coalescing in Graph500 scale 31 graphs when running *DC* SSSP with max edge weight of 100(Figs. 4.4 and 4.7). Figure 4.4 shows the large impact of a small change in the coalescing size, which is measured by the number of SSSP messages per coalescing buffer. Changing the coalescing size by less than 2% causes over 50% increase in the run time. This unexpected effect is caused by the specifics of Cray MPI protocols. At the smaller coalescing size, full message buffers fit into rendezvous R0 protocol that sends messages of up to 512K using one RDMA GET, while the larger buffers hit R1 protocol that sends chunks of 512K using RDMA PUT operations. At the size of 44,000, the bulk of the message fits into the first 512K buffer, and the small remainder

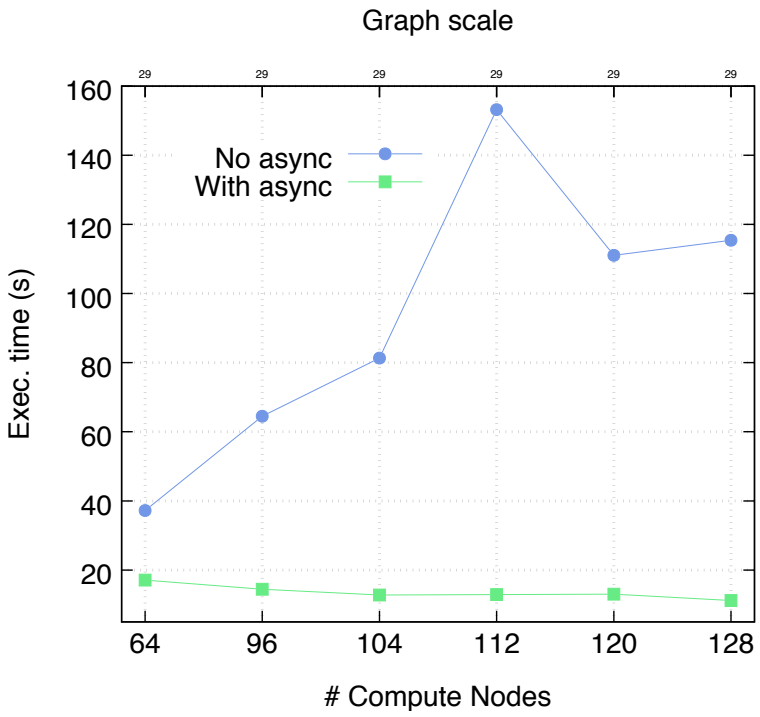


FIGURE 4.8. Effect of asynchronous progress on BR2.

requires another RDMA PUT, causing overheads. The sizes 43,000 and 86,000 fill out 1 and 2 buffers, respectively, achieving similar performance. The larger size, 86,000, results in better scaling properties.

In particular, in our DC-SSSP algorithm, message size is 12 bytes(vertex of 8 bytes and the distance of 4 bytes). Thus, one R1 buffer can store about 43690 messages. When the runtime exceed that size by, say, using 44000, the second buffer sent in the R1 protocol is always very small, causing overhead. The overhead is incurred because R1 is an iterative protocol and for the second smaller buffer it has to go through the same expensive “hand-shaking" protocol as the first buffer of size 43690.

We ran a more extensive suite of benchmarks on Edison. Figure 4.7 shows the coalescing buffer size experiments on Edison. The results are similar, with a periodic increases in the minimum run time as protocol buffers mismatch the coalescing buffers. The maximum run times signify the worst run time, as other parameters related to bit transport than coalescing are adjusted. The results show that adjusting other parameters is less and less important as the coalescing buffer size increases.

Figure 4.5 shows the effects of coalescing on a *DC* BFS, which is SSSP with maximum weight of 1. Surprisingly, increasing the coalescing size impacts performance negatively. We suspect that with smaller weights the possibility of reward from optimistic parallelism in *DC* decreases, and the added latency of coalescing has a much larger effect than with larger weights. All three cases shown in Figs. 4.4, 4.5 and 4.7 show that adjusting the coalescing size is important, and the optimal value is not static. Rather, it depends on algorithmic concerns such as reward from optimistic parallelism.

4.4.3. Transport Progress. At first, when we experimented with *DC* on Big Red 2 with AM++, we found out that it was performing worse than Δ-stepping algorithm [85]. This raised a concern that the *DC* approach may not be practical. We suspected the possibility of message latencies being a culprit; so, upon researching MPT, we decided to experiment with *asynchronous progress*, which uses separate threads to perform progress in certain situations. Here, instead of sending and receiving MPI message buffers using worker threads, we can assign a separate thread to manage buffers, and send/receive MPI buffers. Despite Cray's warning at the time that thread-multiple progress required for asynchronous progress "is not considered a high-performance implementation," we observed significant gains for *DC*, shown in Fig. 4.8. We ran the experiment on Graph500 scale 31 optimal strong scaling results. Without asynchronous progress, performance decreased with the increased number of nodes (with an unexplained anomaly at 112 nodes). (Note that all our experiments are averaged; thus, large anomalies are indicative of unexpected circumstances.) With asynchronous progress thread, the performance of *DC* has improved more than tenfold with growing node counts, entirely changing the viability of the approach. This dramatic effect illustrates how deeply an algorithm interacts with the runtime and how a gap in parameter space may lead to incorrect conclusions about DGA approaches. with asynchronous progress thread, the execution time decreases with the number of nodes because workers are only doing computation. Without an asynchronous progress thread, each worker needs to progress message communication, hence the amount of time spent on computation is reduced. So it is possible that another

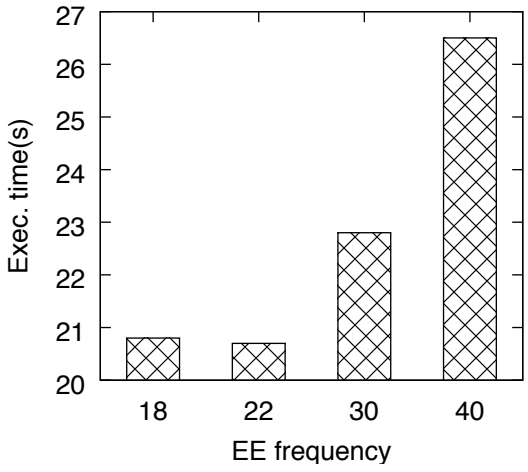
worker will get more sub-optimal works from other workers, when, actually, if the runtime had an asynchronous progress thread, can give workers more chance to propagate good work.

4.4.4. Distributed Control Progress Heuristics. In addition to transport layer progress, AM++ performs its own internal progress when AM++ interfaces are called. Since AM++ *DC* is built around a loop that empties the local priority data structure it must occasionally, with some frequency, call into the appropriate AM++ interfaces that perform progress. This frequency is controlled by two parameters: the end-epoch test frequency (EE) and the eager progress limit (EL).

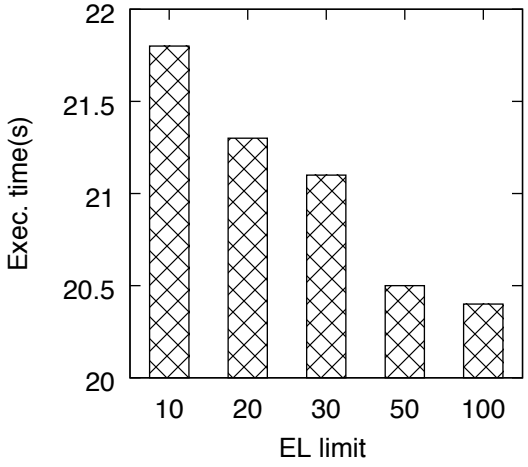
EE controls how many iterations of the *DC* loop run before AM++ progress is invoked. Adjusting this frequency impacts the timeliness of delivery of messages. With higher frequency, AM++ progress is performed more often, and more fresh tasks get into the thread-local priority queues, but the overhead of progress increases. Lower frequency has a thread execute more tasks from its priority queue avoiding the overheads of AM++ progress, but the tasks that are executed are more stale. The best balance between overhead and staleness depends on the structure of the input.

The eager limit is a threshold of outstanding *DC* tasks below which AM++ progress is performed during every iteration of the *DC* loop.

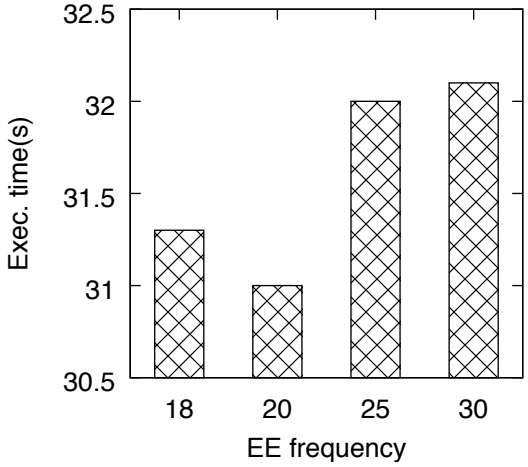
Figure 4.9 shows the effects of progress parameters, using performance data averaged over multiple runs while varying orthogonal parameters and choosing the best performing variant, which isolates the effects EE and EL parameters. Edison shows a significant sensitivity to the EE parameter. Smaller values are better, with 22 being the best of the ones tested. This suggests that latency may be a limiting factor on Edison. On Big Red 2, the results of varying the EE parameter are less pronounced, but the average of multiple experiments that we show here still suggests some sensitivity with the optimal value similar to that on Edison. Altogether, the results show that the performance of *DC* depends on the progress model.



(A) EE on Edison.



(B) EL on Edison.



(C) EE on BR2.

FIGURE 4.9. Effect of AM++ progress parameters.

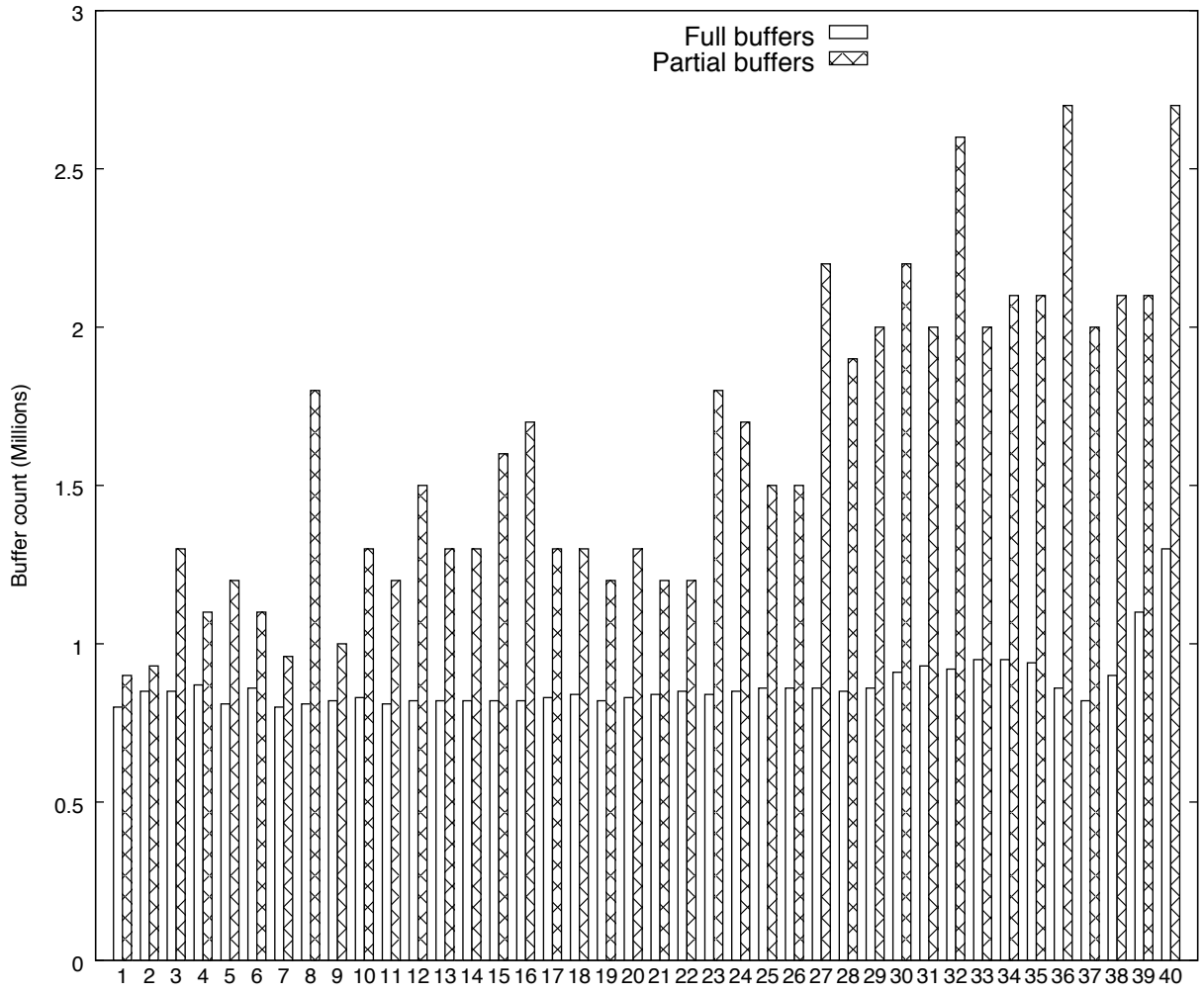


FIGURE 4.10. Partial and full buffer statistics for 40 fastest(fastest to the left, slowest to the right) executions with coalescing size fixed at 100000 (on Edison, across different batch jobs).

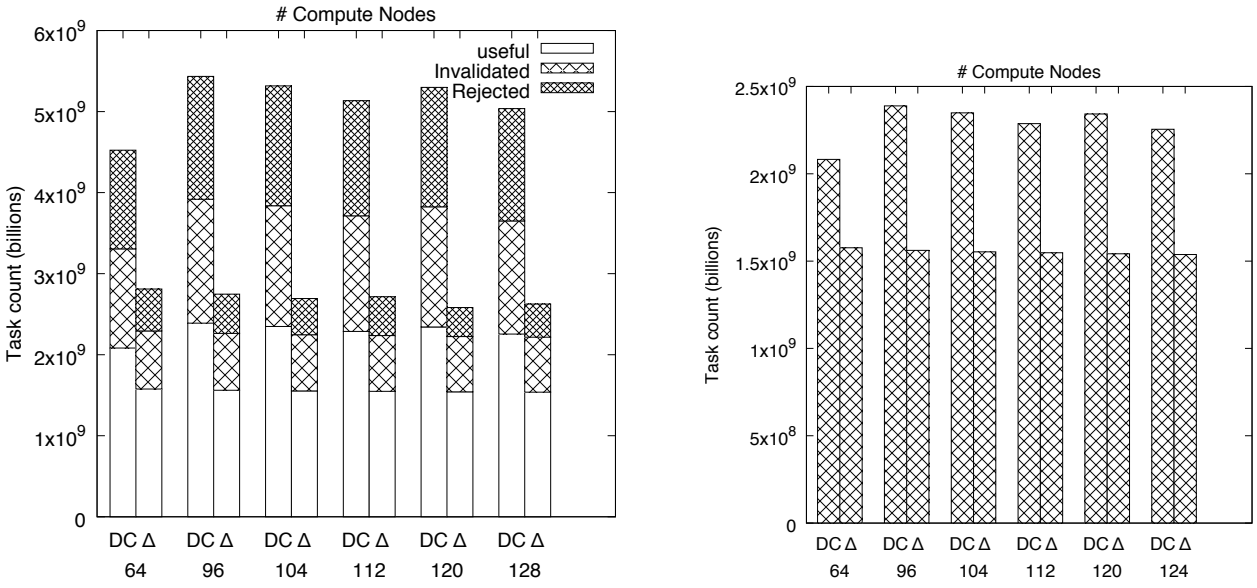


FIGURE 4.11. Work statistics for DC-SSSP and Δ-stepping algorithm in AM++. *(a)* Useful, invalidated, and rejected work. *(b)* Useless work.

4.4.5. Buffering and Work Efficiency. The prerogative of coalescing in AM++ is to decrease the overhead by sending as many full coalescing buffers as possible. Partially filled buffers are only sent when no more messages are being inserted. Figure 4.10 shows *DC* results on Edison for coalescing buffer size of 100,000. We found that the best predictor of performance is the amount of partial buffers (fewer is better) followed by full buffers (more is better). Partial buffers indicate periods of a lack of work, and this, in turn, indicates that the local priority queues are getting depleted more often, decreasing overall performance. AM++ was originally optimized for algorithms like BFS and Δ-stepping, which benefit from eager optimization of communication overhead and are not sensitive to work imbalance. Our example shows that optimization of runtime for a seemingly worthy goal can negatively impact algorithms that have other needs not anticipated by runtime developers.

4.4.6. Work vs. Overhead. Performance of an algorithm depends on the amount of *work* it performs and on the amount of *overhead* that this work incurs in a given runtime. Figure 4.11 shows the work statistics comprising of useful work (vertex distance was

updated), useless work (distance was not updated), rejected work (distance updated but neighbors are not visited) and invalidated work (useful work overwritten by a better distance) for DC and our implementation of Δ-stepping in AM++ with scale 31 graph. Although DC performs better than Δ-stepping, DC always executes more work than Δ-stepping in the most efficient configurations of both of the algorithms. Despite consistently performing 10%-25% more work, DC performs better in all instances of tests at scale (3-6 times speedup). This shows that synchronization and uneven distribution of work have an important effect on the performance of DGAs. Although one can attempt to mitigate the work imbalance with algorithmic techniques, the cost of synchronization is hard to control and eliminate. In this regard, an underlying runtime can have a significant impact. The more an algorithm depends on keeping global information about the runtime (e.g., for load balancing), the higher the costs of synchronization necessary to maintain that information. In Figure 4.11 we count a task as rejected when the vertex distance it delivers is higher than what is already recorded and, consequently, the task is not inserted into the priority queue of DC or a bucket of Δ-stepping. Invalidated tasks are similar to rejected tasks, but their distance expires while they wait in priority queue.

4.5. Conclusions

We demonstrate that the algorithm-level parts of DGAs that are reported as major contributions do not constitute a complete description of a DGA. A DGA consists of two equally important layers: the algorithm-level aspects and the runtime-level aspects, which respectively represent the top and the bottom of the software/hardware stack. Based on analysis of a representative sample of DGAs, we further subdivide the layers into categories. We propose a template for reporting research design and results and we demonstrate how to use it. Altogether, the goal is to make research results in DGAs more accessible, general, and congruent.

Our Tables 1 and 2 serve as a map for reporting the design features and the related quantities relevant for interpretability of experiments. Some runtime aspects may remain "buried in the stack", their impact unknown (e.g., the effects of job placement as in Sec. 4.2.1.2 are not usually investigated), and some may not be relevant in a given situation. A complete "report card" helps one understand which parts of the parameter space are covered and which are not. Our reporting template helps both consumers and authors of research, the former to understand and the latter to present contributions.

Our analysis and guidelines are the first step in unifying the field. We posit that the DGA research community should collectively develop a set of standards expected from top notch research, acknowledging that DGAs exhibit particularly strong interaction with the software/hardware stack due to their irregularity. Thus we appeal to the wider community to help develop standards for more explicit incorporation of runtime interactions in future research results and by collaboration on a continuously updated consensus on what constitutes the runtime of a DGA.

CHAPTER 5

Runtime Scheduling Policies for Synchronization-avoiding Graph Algorithms

In this chapter, we explore scheduling and runtime system support for unordered distributed graph computations that rely on optimistic (speculative) execution. Performance of such algorithms is impacted by two competing trends: the higher degree of parallelism enabled by optimistic execution in turn requires substantial runtime support. To address the potentially high overhead and scheduling complexity introduced by the runtime, we investigate customizable scheduling policies that augment the scheduler of the underlying runtime to adapt it to a specific graph application. We present several implementations of our synchronization-avoiding graph algorithms, also termed as Distributed Control (*DC*) , a data-driven unordered approach with work prioritization and demonstrate that customizable scheduling policies result in the most efficient implementation, outperforming the well-known Δ-stepping Single-Source Shortest Paths (SSSP) and Jones-Plassmann vertex-coloring algorithms. We apply two scheduling techniques, *flow control* and *adaptive frequency of network progress*, which allow application-level control over the balance of domain work and the runtime work. Experimental results show the benefit of such application-aware scheduling for irregular distributed graph algorithms.

5.1. Introduction

Adapting runtime scheduling policies for *regular* and *numerical* kernels (for example BLAS, FFT) is a well-studied area [19, 21, 12]. However, interaction between *irregular* data-driven applications and the underlying *runtime scheduler* has not been studied extensively beyond shared memory systems [92]. Yet these applications are demanded for processing big data sets arising from many contemporary problems of interest, e.g., social graphs.

Irregular applications such as graph algorithms differ from regular and numerical kernels: they exhibit little locality, rarely require any significant computation per memory access, and result in high-rate communication of small messages. In graph applications, work items are generated in an unpredictable pattern. This execution-time behavior of graph algorithms makes their performance heavily dependent on the whole software/hardware stack, which includes not just the algorithm itself but all levels of the runtime [50]. Traditional bulk-synchronous parallel (BSP) graph algorithms such as Δ-stepping single-source shortest paths (SSSP) [85] and Jones-Plassmann [66] coloring algorithms employ global and vertex-centric barriers respectively.

Unordered algorithms [113, 98] maximize available parallelism through *optimistic parallelization* [70] and enable asynchrony by executing independent computations concurrently. Because dependencies are not checked a priori, the results computed previously may need to be corrected. These *label-correcting* mechanisms have the advantage of avoiding synchronization and the straggler effect [113]. However, there is a consequence to this kind of speculative parallelization. Speculation and parallelism need to be carefully balanced to provide enough parallel work while avoiding excessive correction.

The balancing of speculative execution and correction of sub-optimal work depends on how the runtime system schedules application work and communication. A runtime system, being general, has no knowledge of the algorithm and semantics of the data. Graph algorithms are not the common case and differ from one algorithm to the next, from one graph type to another. Striking the balance between speculation and communication has to be done at the level of the graph algorithm, however communication and scheduling are aspects of the runtime. To bridge that gap, graph algorithms need runtime hooks to influence scheduling policies.

In this work, we show that to maximize performance of unordered graph algorithms it is imperative to employ pertinent *scheduling policies*. Furthermore, we show that it is vital that these scheduling policies are *application-aware*, i.e., driven by the characteristics of the application. We devise the *adaptive frequency* and *flow control* techniques to affect

scheduling of internal runtime tasks such as network progress. Adaptive frequency regulates the frequency with which the runtime performs communication progress, based on the balance of application and progress work. Flow control ensures that the application does not progress too far locally without sufficient global progress in the state of the application. Excessive local progress may generate waves of wasted application work. Graph applications are particularly vulnerable to unbalanced progress because local work is usually cheap while communication (memory and remote) incurs a latency penalty. However, the issues we tackle are common to any optimistically parallel applications that exploit available parallelism.

We study a family of unordered algorithms [98] for SSSP and coloring on a more sophisticated asynchronous many-task (AMT) runtime named HPX-5, based on an earlier implementation of synchronization-avoiding SSSP [113] in AM++ runtime, and we demonstrate the effectiveness of application-driven scheduling for graph computation. We analyze the performance of *DC* implementations using different scheduling policies and one that uses the default scheduler, and we compare them against the well-known Δ-stepping SSSP algorithm [85] and Jones-Plassmann coloring algorithm [66]. Our results show that unordered algorithms can perform better than their ordered counterparts when empowered with adequate runtime support. In summary, the contributions we make include:

- For *distributed graph applications*, we investigate the interaction between algorithms and plug-in runtime scheduling policies that augment the main scheduler of an asynchronous many-task (AMT) runtime system. Such customized policies can control application-level scheduling (priority scheduler in our case) and low-level runtime functions such as network progress.
- We show that, with proper runtime support, unordered algorithms can perform well at scale. Our implementations of *DC* outperform the baseline Δ-stepping SSSP algorithm and Jones-Plassmann coloring algorithm, when given an adequate control over runtime scheduling.

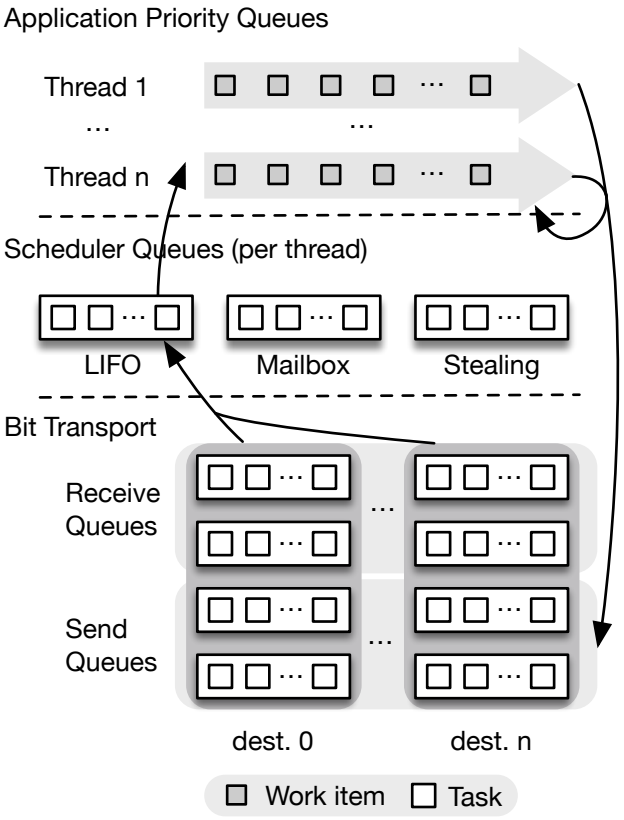


FIGURE 5.1. A simplified diagram of the placement of and of the interaction between application and runtime-level queues.

- We identify two application-level techniques: flow control and adaptive frequency of network progress, that make unordered algorithms perform better. Our analysis and techniques can be applied to other AMTs and other unordered algorithms.

5.2. The Case for Custom Runtime Scheduler

In the following discussion, we distinguish the unit of work executed by the runtime into two separate categories: application-level work items (such as updating a vertex property) and runtime tasks (such as network progress, termination detection etc.).

Our approach for formulating synchronization-avoiding graph algorithms, also named as distributed control approach (*DC*) [113], is a data-driven, unordered, label-correcting algorithmic approach based on speculative execution. In DC algorithms, a vertex updates its information (distance, color, etc.) whenever a better value is received from one of its neighbors. The propagation of messages from the neighbors is completely asynchronous.

The DC approach can execute work in no particular order and refines the result as the algorithm progresses. This flexibility in ordering facilitates parallel computation without global synchronization (compared to Δ-Stepping) or vertex-centric synchronization (compared to JP). Since execution of both of these *DC* algorithms does not depend on any order of work items (unordered), this may result in executing sub-optimal work in intermediate steps.

To reduce the amount of sub-optimal work, *DC* performs local ordering of work items in the thread-local priority queues (application-level queues in Fig. 5.1). However, the global work order obtained by local prioritization is strongly influenced by the low-level execution order in the distributed runtime. This necessitates runtime support for quick delivery of messages from the bit transport layer to the application layer, so that they can be ordered as soon as possible. However, a runtime system usually consists of several layers of abstractions, with tasks and work items scattered across different layers (Fig. 5.1). For example, a work item can be in the local queue of a runtime waiting to be scheduled for execution, in a thread-level priority queue in the application level, or in the bit-transport layer buffers. In this regard, we have made a couple of observations on the interaction between the underlying default runtime scheduler and unordered algorithms such as *DC*. The following discussion is based on the HPX-5 runtime, but in general, most distributed AMT runtimes have similar software stack as shown in Fig. 5.1 and follow the same set of steps in the runtime scheduler as shown in Alg. 11. Consequently, unordered graph algorithms would face similar challenges on these runtimes.

The HPX-5 runtime system is an initial implementation of the ParalleX model [52] and a good representative of AMT runtimes. HPX-5 represents work as *parcels*, a form of active messages. The HPX-5 runtime scheduler is responsible for executing actions associated with parcels. It is a multi-threaded, cooperative, work-stealing thread scheduler, where heavy-weight worker threads run scheduler loops that select parcels to be executed. The scheduler loop is outlined in Alg. 11. Every HPX-5 worker thread running the scheduler keeps spinning until it finds a parcel to execute or it has been signaled to stop. Specifically,

each worker thread, t_{id}, in HPX-5 maintains a last-in-first-out (LIFO) queue, L_{tid}, of parcels (Ln. 9), with a possibility of stealing the oldest parcels from other threads. The light-weight threads executing parcels can yield, and HPX-5 maintains separate queue, Y_q for yielded threads (Ln. 6). Parcels can be sent to particular heavy-weight scheduler threads using mail queues, M_{tid} (Ln. 3). Newly generated parcels may be destined for remote localities (ranks), and HPX-5 provides transparent bit-transport layer with robust implementation based on the Photon [69] RDMA library and an implementation based on Message Passing Interface (MPI).

In the scheduler, mailboxes are given the highest priority, followed by the yield queue, followed by the LIFO queue. Next, a plugin-scheduler, an extension we discuss in more detail in Sec. 5.3 gets a chance to execute. Finally, when the scheduler is unable to obtain work from thread-local sources, it first attempts to progress the network (retrieving messages from the receive queues, N_r, in bit transport in Ln. 14) and then to steal work from other scheduler threads. It is important to note that executing work in any of the steps causes the loop to start from the beginning. So, for example, all mail tasks will be processed before any LIFO queue tasks, and no network progress will be performed before all work sources that come before it in the scheduler loop are exhausted. While this domain-demand-oblivious fixed-step scheduling approach works well for some applications, this leaves little room for interleaving domain work and network progress, which is essential for obtaining local ordering to reduce sub-optimal work for algorithms such as *DC* that depend on efficient scheduling.

At a particular instance of time, the scheduler needs to make a trade-off between executing a task or a work item. If the balance is chosen poorly, work can get stuck in the network buffers while the scheduler tries to execute parcels from the application-level priority queue. This can leave the *DC* algorithms with little choice to compare and choose from a smaller number of work items in the application-level per-thread priority queues. This results in dwindling priority queues used for local ordering in *DC*, even if work items are available in the transport buffers.

Alg. 11: Runtime scheduler loop

In : Plugin algorithm-level scheduler A_s with a work produce f_p function

```
1  while A task t_i or a work item w_i available do
2      if M_tid ≠ ∅                                  ▷ Mailbox queue (per thread)
3      then
4          Execute t_i ← M_tid.pop() and continue;
5      else if Y_q ≠ ∅                               ▷ Yield queue (per process)
6      then
7          Execute t_i ← Y_q.pop() and continue;
8      else if L_tid ≠ ∅                             ▷ LIFO queue (per thread)
9      then
10         Execute t_i ← L_tid.pop() and continue;
11     else if w_i ← f_p not NULL then
12         Execute w_i and continue                  ▷ Plugin scheduler
13     else if N_r ≠ ∅                               ▷ Network receive queue
14     then
15         L_tid.enqueue(N_r) and continue;
16     else
17         Try work stealing from L_tid' and continue   ▷ Steal from another
               thread tid'
```

Based on these observations, we posit that distinguishing runtime tasks from algorithmic work items to facilitate scheduling and having a way to provide a graph algorithm-specific scheduling policy in the runtime scheduler can benefit unordered algorithms in several ways. First, by separating the consideration given to these sets of works, the runtime has better control over when to schedule what type of work. Second, the runtime can capitalize on programmer's knowledge about algorithmic work items. For example, the application programmer can provide an ordering policy for the work items (for example, priority for parcels containing shorter distances). Third, we note that graph algorithms are communication-bound, rather than computation-intensive. If, at any particular time instance, the application level does not have enough work items to work on or compare with, it can voluntarily give up control to other scheduling mechanisms such as network progress to fetch more work from the underlying transport. Delaying network progression till exhaustion of work items eliminates the chance of propagating better work from other ranks sooner. Such interleaved execution of runtime tasks, work items, and network progress can boost the performance of an unordered algorithm.

A runtime scheduler matches the granularity of the expression of an algorithm and its realization in the runtime system. However, the small units of computation of fine-grained vertex-centric algorithms pose challenge on balancing between execution of application work and progressing the runtime itself. Consequently, runtime intervention is needed to support lower-level optimizations such as message aggregation on the sender side to coarsen the computation. Once messages arrive at the destination, the computation resumes the fine-grained execution. The dynamic and unpredictable remote memory access pattern puts the scheduler on the critical path of performance of unordered speculative graph algorithms. To alleviate these issues, we extend the general-purpose scheduler with configurable plug-in scheduling. To provide appropriate scheduling on application-specific basis, the plug-in mechanism is needed so that the application can provide best scheduling policies.

The scheduling policies described in the next section are implemented to provide feedback to the runtime based on the information and hints available at the application level such as the growth of thread-local algorithm-level priority queues to decide when to progress the network as well as application-level parameters to indirectly throttle sending of messages. None of these techniques are encoded into the runtime, and therefore such application-level policies can be implemented for any runtime.

5.3. Scheduling policies for Synchronization-avoiding Graph Algorithms

Asynchronous execution of unordered graph algorithms eliminates global and vertex-centric barriers. Time saved by eliminating such synchronizations can be invested in speculative execution. However, speculative execution results in wasted work, the amount of it depending on the quality of speculation. Thus there is a break-even point between synchronization overhead and amount of sub-optimal work to execute. Two relevant aspects of an underlying runtime impact this choice: interleaving of different tasks and communication over the network. To this end, the runtime system needs to regulate the back-pressure (flow control) and increase-decrease the network progress as needed (adaptive frequency). In the following, we first give a high-level description of how a runtime can incorporate such techniques for unordered graph algorithms and then discuss HPX-5 runtime-specific way of implementing such techniques.

Since graph algorithms are mostly data-driven, messages are generated in an irregular fashion and are targeted to random remote localities (ranks). Unordered algorithms are more susceptible to irregular nature of message propagation because of the assumption that sub-optimal results will be corrected once better values become available. This can result in overwhelming the remote receivers, if the amount of available work is not throttled to acceptable levels. Hence, it is necessary to regulate the "back-pressure" of messages to adjust work production rate to work consumption rate. The regulation mechanism can be implemented locally, on the sender side, or by getting feedback from the sender's communication layer, or by communicating with the remote receiver. Controlling the

flow of messages by communicating with the remote target can be done either frequently (heavyweight) or infrequently (lightweight). Although *heavyweight flow control* can give a better idea about the precise state of the entire system, it incurs significant overhead. On the other hand, *lightweight flow control* can provide approximation of the receiver's responsiveness with low overhead. We have implemented a lightweight flow control mechanism to adjust back-pressure properly. Moreover, lightweight flow control can be done in different granularities. To get an idea about how much work has been done on the receiver side, we can insert "beacons" in the work stream that notify the sender that they have been executed. Beacons can be inserted either randomly (coarse-grained) or in a per-destination (fine-grained) fashion. Fine-grained "beaconing" requires heavy book-keeping with data structures and computation to keep track of beacons. Coarse-grained low-precision beaconing mechanism, on the other hand, gives a good approximation of unbalanced overwhelmed receivers in distributed setting with statistically high probability.

Beaconing helps the sender decide if more work should be sent, or if more resources should be devoted to progressing the runtime. Receiver, on the other hand, has to balance the resources put into runtime progress and application-level work to avoid the risk of running into having too many messages stuck in the low-level transport queues. Too many messages waiting in buffers can lead to depleted priority queues execution of excessive sub-optimal work. Too much effort put into network progress can interfere with useful work. To mitigate that tension, network *progress frequency* needs to be adjusted to achieve the right balance between these tasks. In Secs. 5.3.1 and 5.3.2 we discuss how we implement flow control and adaptive network frequency mechanism in HPX-5.

Alg. 12 and Alg. 13 show the pseudo code for the *DC* approach with flow control and progress frequency heuristics. The algorithm consists of 3 parts: the work produce function f_p that manages extraction of algorithmic work items from the local priority queue, the message handler that receives tasks from other workers, and the relax function that updates properties and generates new work. The basic task of f_p is to remove work items from the thread-level priority queue (Ln. 11 in Alg. 12) (such as vertex-distance pair

Alg. 12: DC with Adaptive Frequency and Flow Control

```
1  procedure Work produce, f_p
2      if sync_count == sync_threshold then
3          return NULL    ▷ Outstanding synchronous calls reached the
                            threshold
4      else
5          if not q_tid.empty() and q_tid.size() > last_size then
6              freq[tid]--        ▷ Process work from priority queue less
                                    frequently
7          else
8              freq[tid]++        ▷ Process work from priority queue more
                                    frequently
9          last_size ← q_tid.size()
10         if not q.empty() and processed[tid]-- > 0 then
11             (v, d) ← q_tid.pop()                ▷ next work item to process
12             return w_i ← (v, p)
13         else
14                 ▷ Reset the number of work items to be processed in
                     the next iteration
15             processed[tid] = freq[tid]
16             return NULL
17 procedure Receive handler
       In  : Work item (v, p)
18     q_tid.push(v, p)              ▷ insert work item into priority queue
```

Alg. 13: Relax function with flow control mechanism

```
1  procedure Relax
       In  : Work item (v, p), Propertymap P
2      if Constraints are satisfied then
3          P(v) ← p
4          for v_n ∈ neighbors(G, v) : do
5              if send_count[tid] < send_threshold then
6                  send_count[tid]−−
7                  send_async((v_n, f(P(v), edgeproperty(v, v_n))))
8              else
9                  sync_count++
10                 send_async_with_cont((v_n, f(P(v), edgeproperty(v, v_n))),
                     λ.sync_count−−)
11                 send_count[tid] = send_threshold
```

for SSSP or vertex-colorinfo pair for coloring) and to return them to the runtime scheduler for execution. The runtime scheduler then runs the relax function (Alg. 13). This function first checks whether the work item contains better distance or color information (Ln. 2 in Alg. 13) and if so updates the receiver's color or distance (Ln. 3 in Alg. 13). Finally it sends updates to all neighbors of the vertex being relaxed.

5.3.1. Flow Control. Local ordering in *DC* produces better optimal work ordering when more work is available to order in thread-local priority queues. The runtime, however, needs to deliver messages across the network through multiple layers of implementation. This causes a tension between *DC* and the runtime, where, on the one hand, it is best to deliver majority of work items into *DC* priority queues, but, on the other hand, minimizing the amount of work items that are in-flight in the runtime comes at a cost of

runtime overhead. We implement a flow control mechanism to allow *DC* to control the flow of network communication through the HPX-5 runtime using customizable parameters.

Work items are moved out from the network layers of HPX-5 when the scheduler loop in Alg. 11 runs network progress (Ln. 14). The only way that control reaches Ln. 14 is when the work produce function returns a null work item (Ln. 11 in Alg. 11). Our plugin-scheduler *DC* maintains an approximate measure of work items that have been sent over the network but not yet delivered. To maintain the approximation, we keep a per-rank global counter $sync\_count$ of work items that have been sent with a request of remote completion notification. When this count grows over some threshold $sync\_threshold$, f_p returns control back to the runtime (Ln. 3 of Alg. 12).

In the *Relax* function (Alg. 13), when the worker thread propagates updated distance to the neighbors (Ln. 4 in Alg. 13), it checks how many asynchronous sends have been posted (Ln. 5 in Alg. 13). If the count has reached a particular threshold $send\_threshold$, a send with continuation (beacon) is performed and the $sync\_count$ value is incremented to keep track of how many continuations are expected (Lns. 9–10 in Alg. 13). When calls with continuation are completed remotely, the continuation decrements the $sync\_count$ value on the rank from which the original send call was made. At every send with continuation, the thread-local $send\_count$ is reset to 0. The call with continuation is performed with the `hpx_call_with_continuation` HPX-5 function:

```
1  hpx_call_with_continuation( addr, action, c_target, c_action, ...)
```

`hpx_call_with_continuation` takes an address *addr* (local or remote) and invokes the specified action *action* at that address. Once that action has finished executing, the continuation action *c_action* is invoked at *c_target* address. The continuation is "fire and forget," and it is automatically handled by the runtime.

5.3.2. Adapting Frequency of Network Progress. If the current rank (locality) keeps receiving messages and the network progress keeps succeeding with adequate amount of work items received over the network, it is an indication that either the algorithm is in the

middle of its execution phase or a lot of messages are destined to the current rank. It is thus useful to keep retrieving messages from the network receive buffer and put them in the priority queues in the algorithm level. In this way, when the algorithm gets a chance to progress, it has robust amount of work items in the priority queue to compare and make choices from and minimize the possibility of executing sub-optimal work items.

To get an idea of successful network progression, the algorithm checks the current priority queue size in the f_p function and compares it with the size seen the last time. Growing size of the priority queue is an indication of successful network probing (Ln. 5 in Alg. 12). As mentioned earlier, it is better to fetch more work items from the network aggressively if the network progression keeps returning a lot of received messages. To achieve this, the algorithm maintains a thread-local counter *freq*. Whenever the queue size grows, the *freq* counter is decremented to indicate that fewer elements will be processed from the priority queue and control will be given to the scheduler to progress the network more frequently (Ln. 6 in *Work produce*).

It is noteworthy to mention here that progressing the network for every vertex processed is not a viable option. The reason is that network progress incurs much more overhead compared to processing a vertex. Although eager network progress can assist in the reduction of useless work by increasing priority queues' size, it has detrimental effect on algorithm performance due to the associated overhead.

5.4. Experimental Results

In this section, we evaluate several algorithms based on *DC* and compare their performance with the baseline Δ-stepping SSSP and Jones-Plassmann coloring algorithms. First we present our analysis in detail based on SSSP algorithms. Next, we show relevant important results for coloring algorithms in Sec. 5.4.4. This additional graph application provides supporting evidence that regulating message flow and adapting network progress are general techniques applicable to different algorithms.

In the following discussion, algorithms without plugin scheduler carry np subscript, algorithms which give up control to the runtime scheduler at a fixed frequency carry ff subscript, algorithms with flow control carry fc subscript, and algorithms with adaptive frequency for network progress carry af subscript.

5.4.1. Experimental Setup. For input, we used Graph500 Kronecker graphs [56], the real world full USA road network and Random4-n expander families of graphs from 9th DIMACS implementation challenge [4]. Each family of graphs has different structural properties. For example, Random graphs have shorter paths (expected depth $\theta(log\ n)$) but demonstrates poor locality. For Graph500 input, a graph of scale n designates a graph with 2^n vertices and an average vertex degree of 32 (edges are considered bi-directional). The full USA road network has 23,947,347 vertices and 58,333,344 edges in total. For Random4-n family of graphs, for a scale n graph input, total number of vertices and edges are 2^n and $4 * 2^n$ respectively, and weights are in the range $[0..2^n]$. For each algorithm, we run 4 problem instances (starting at different sources) and report the average of the execution time with the standard deviation of the mean as the measurement for uncertainty. For each types of graph input, we chose the appropriate Δ value based on the structural properties (e.g., diameter, weight distribution) of a particular graph type. In particular, we set delta to 100000 and 2000 for Random4n and road network. With Graph500 input, we have found $\Delta = 1$ to be the best for Δ-Stepping algorithm. The graph is distributed across different nodes in 1D fashion and represented with a distributed adjacency list data structure. We have compiled our code with GCC 6.2 and with optimization level -O3. Additionally, single node experiments were performed with networking turned on. We used Photon put with completion (PWC) [69] messaging transport layer of HPX-5 runtime for our experiments. We conducted our experiments on a Cray XC40 system and on our own cluster Cutter. Compute nodes on the XC40 system have two different processor configurations: 32 2.7 GHz or 44 2.2 GHz cores, with 64 GB and 128 GB of memory respectively. All results except for the results in Fig. 5.2 and Fig. 5.7 were obtained on the system with 44 cores. All XC40 compute nodes are connected through the Cray

Aries interconnect.The Cutter cluster consists of 16 nodes with 16 Haswell cores at 2.6 GHz each and is equipped with InfiniPath and Mellanox interconnects.

5.4.2. Comparison of SSSP Algorithms With Different Scheduling Policies. We observe that flow control mechanism has more effect on performance compared to adaptive frequency. Figure 5.2 shows the execution time taken by different SSSP algorithms. DC, which uses the plugin capability but does not have flow control or adaptive frequency technique performs worse than DC_{np}. Adding a fixed frequency technique for network progression helped DC_{ff} to perform comparatively up to 8 compute nodes but for larger scale the performance of DC_{ff} deteriorates. Although for smaller scales fixed frequency technique is good enough, to achieve better scaling, the algorithm needs to adjust the network probing according to the work profile, which we do in DC_{af}. Compared to DC_{ff}, this technique worked better with scale 24 graph input but did not perform well with scale 25 input. In $DC_{ff,fc}$, we add flow control. Flow control mechanism helps $DC_{ff,fc}$ in achieving almost identical performance as Δ-stepping algorithm. Finally, $DC_{af,fc}$ performs the best. Flow control and adaptive frequency together make $DC_{af,fc}$ achieve better work ordering and balance in executing tasks and work items.

Figure 5.2 also shows the work profiles for different SSSP algorithms. Here activity count implies total amount of work executed by each algorithm. Although Δ-Stepping executes less amount of work, it has longer execution time. On the other hand, $DC_{af,fc}$ algorithm executes more work due to speculative execution of sub-optimal work but still runs faster. This is due to the fact that, with proper flow control and adaptive frequency technique, instead of waiting on barriers, $DC_{af,fc}$ can schedule work items efficiently and interleave runtime progress and work item execution in a proper manner.

5.4.3. SSSP Scaling Results. In this subsection, we discuss the strong and the weak scaling behavior of $DC_{af,fc}$ and Δ-stepping algorithms with different kinds of graph input at larger scale.

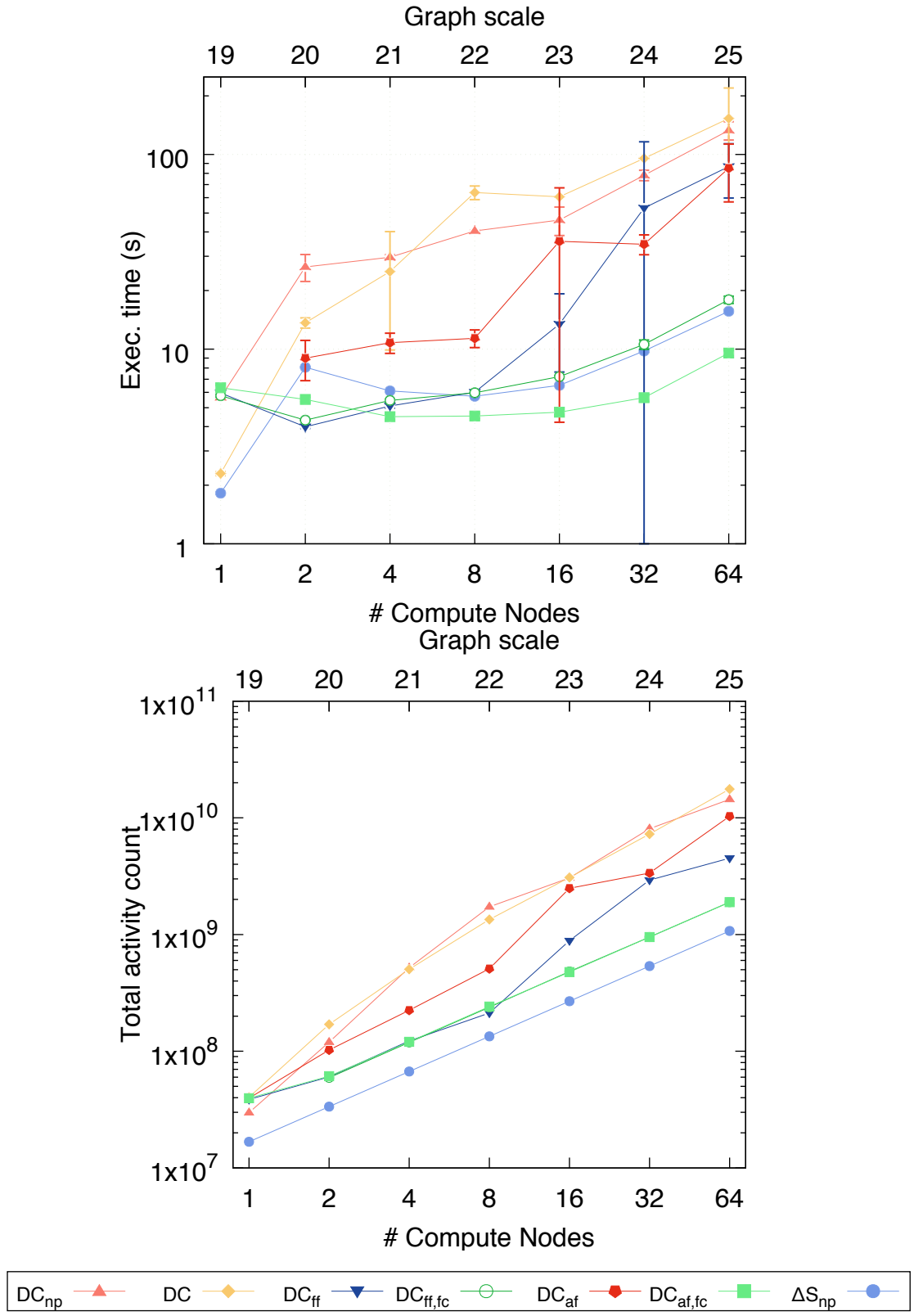


FIGURE 5.2. Weak scaling performance and work statistics of SSSP algorithms with Graph500 input.

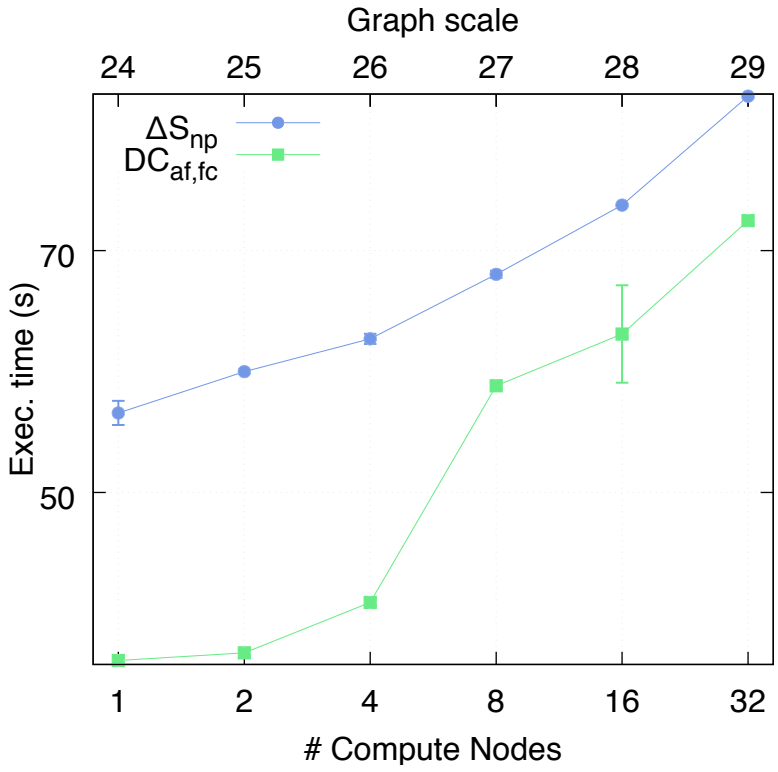


FIGURE 5.3. Weak scaling result of $DC_{af,fc}$ SSSP and Δ-stepping algorithms with Graph500 input.

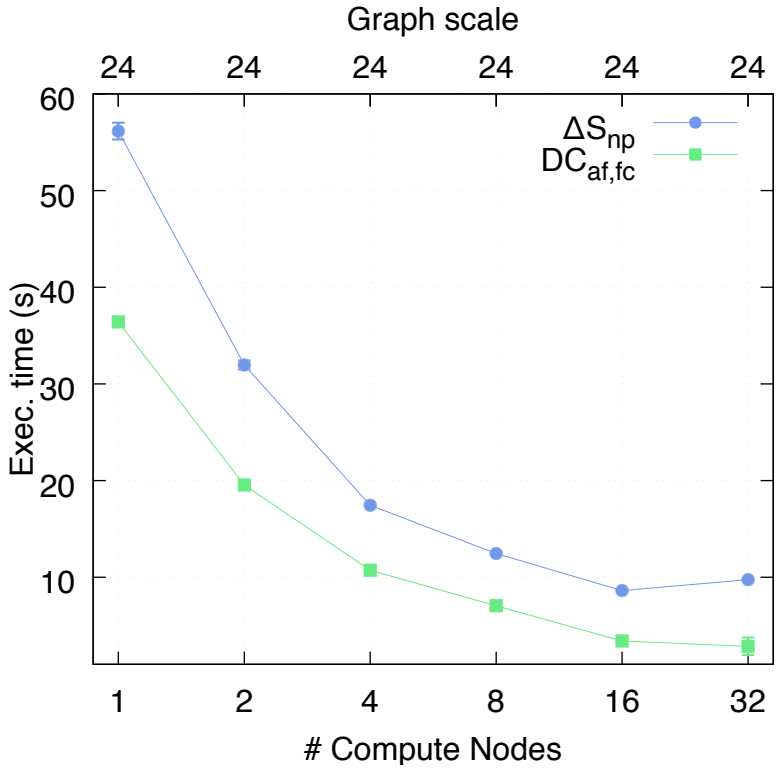


FIGURE 5.4. Strong scaling result of $DC_{af,fc}$ SSSP and Δ-stepping with Graph500 input.

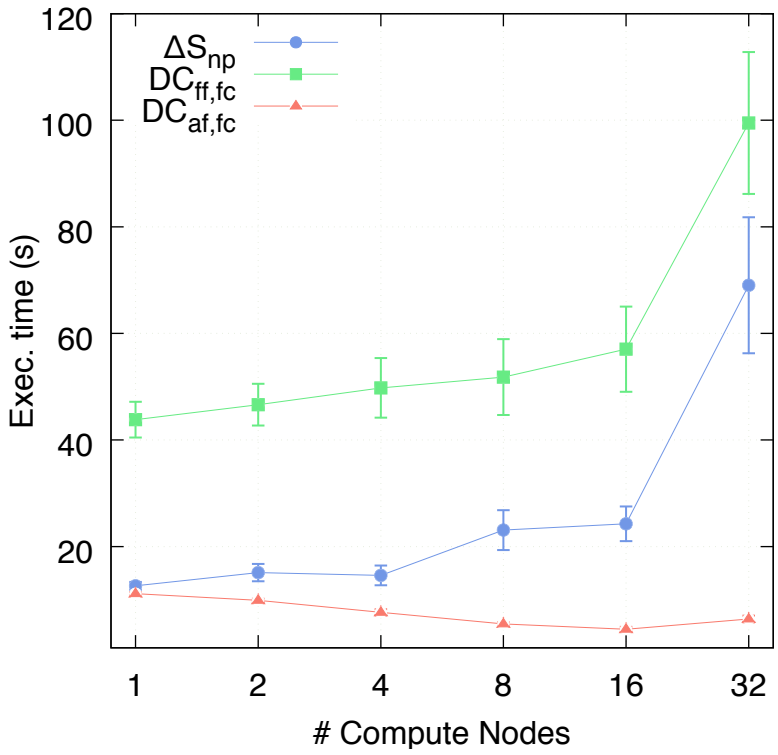


FIGURE 5.5. Strong scaling result of $DC_{af,fc}$, $DC_{ff,fc}$ SSSP and Δ-stepping $_{np}$ with full USA road network.

5.4.3.1. *Graph500 Weak and Strong Scaling.* We demonstrate the weak and the strong scaling behavior of $DC_{af,fc}$ and Δ-stepping algorithms with larger Graph500 input in Figs. 5.3 and 5.4 respectively. In both cases, $DC_{af,fc}$ performs better than Δ-stepping. As shown in Fig. 5.4, both algorithms achieve better execution time with additional compute nodes. Beyond 16 nodes, the execution time of $DC_{af,fc}$ almost flattens while Δ-stepping algorithm experiences a slight increase in execution time with 32 compute nodes.

5.4.3.2. *USA Road Network Strong Scaling.* Figure 5.5 presents strong scaling behavior of $DC_{af,fc}$, $DC_{ff,fc}$ and Δ-stepping $_{np}$ with the USA road network. As can be seen from the figure, the fixed frequency policy for giving up control to the scheduler does not yield good performance in $DC_{ff,fc}$ algorithm. Although with smaller number of compute nodes, Δ-Stepping performs comparably with $DC_{ff,fc}$, the global synchronization barrier quickly becomes a bottleneck. Adaptive frequency along with flow control help $DC_{ff,fc}$ to get the best performance.

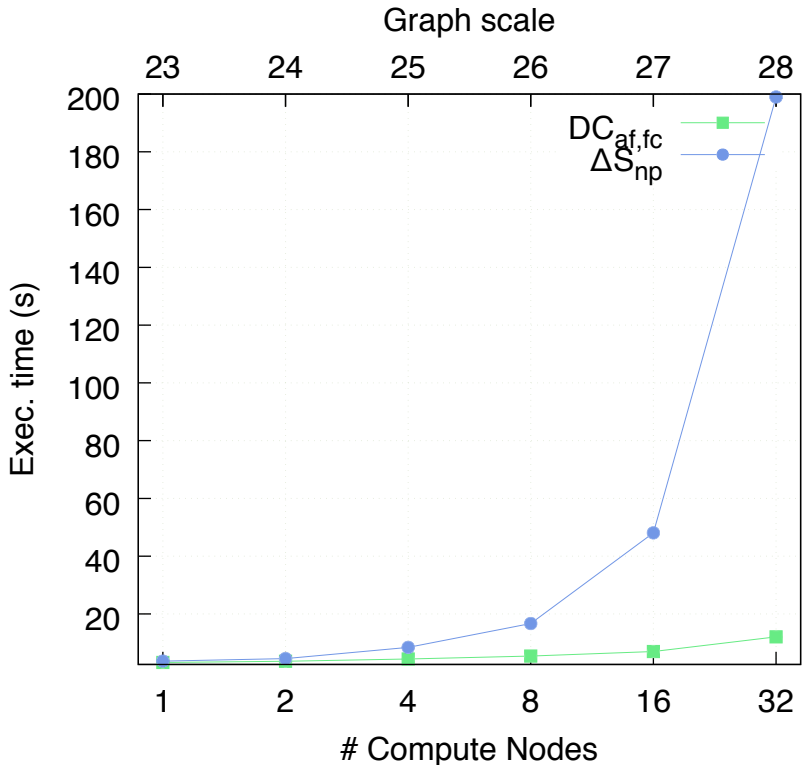


FIGURE 5.6. Weak scaling result of $DC_{af,fc}$ SSSP and Δ-stepping with Random4-n graph input.

5.4.3.3. *Random4-n Weak Scaling*. Figure 5.6 shows the weak scaling results for $DC_{af,fc}$ and Δ-stepping algorithms with Random4-n graph input. Here $DC_{af,fc}$ also accomplishes better performance at higher scales.

5.4.4. Graph Coloring. To demonstrate the applicability of our scheduling techniques to other graph applications, we experiment with per-vertex counter based *DC* coloring algorithm and incrementally add different features. As can be seen from Fig. 5.7, *DC* coloring algorithm with default runtime scheduler performs comparably with Jones-Plassmann (JP) algorithm, but due to work explosion beyond scale 13, it can not finish execution in a reasonable time. Adding adaptive frequency of network progress feature to *DC* helps the algorithm to discard some sub-optimal work. Nonetheless, the execution time still grows uncontrollably with scale. When *DC* is equipped with both flow control and adaptive frequency, the workload becomes manageable and we see the benefit of optimistic parallelization. Figure 5.8 presents weak scaling results for coloring algorithms

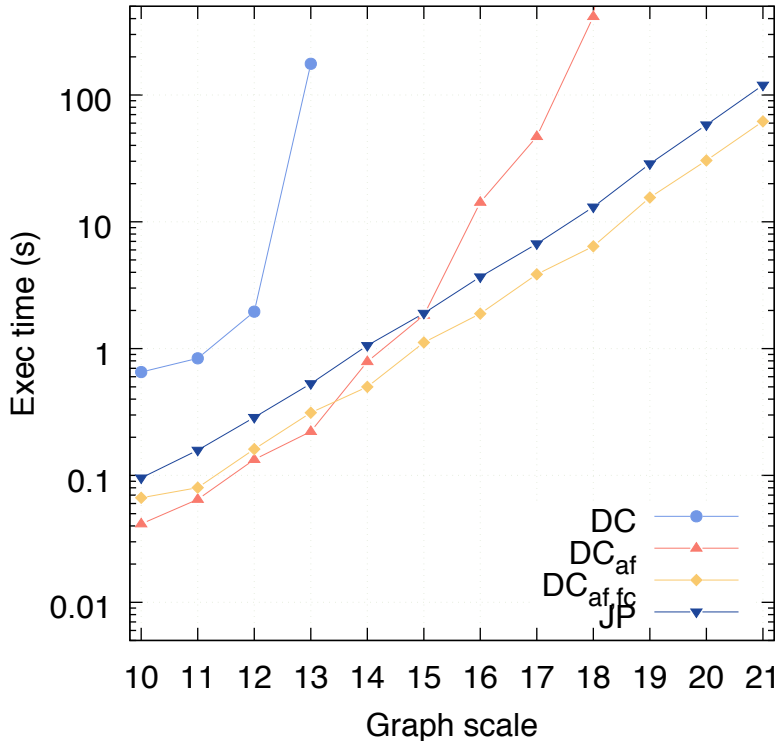


FIGURE 5.7. Performance of coloring algorithms with Graph500 on 2 Cutter nodes (log scale execution time). All the algorithms achieve same color quality, hence total color count is omitted.

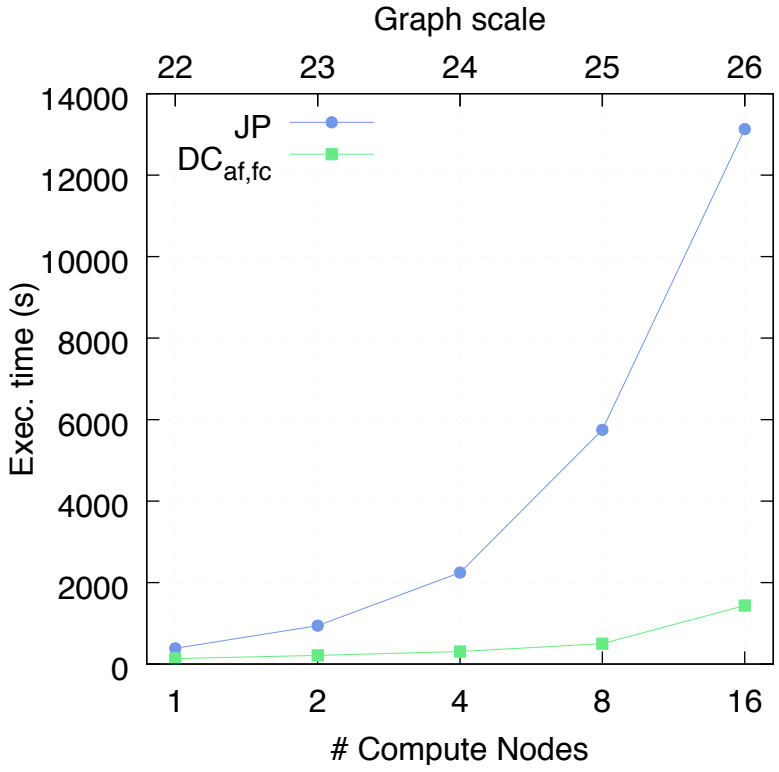


FIGURE 5.8. Weak scaling results of coloring algorithms with Graph500 input.

Send threshold	50	100	200	2000	4000	10000	20000	500	700	1000
Sync threshold	25	50	100	10	5	2	1	250	350	500
Activity count (in billions)	1.893	1.894	1.898	1.899	1.901	1.903	1.904	1.916	1.928	1.976
Time (s)	21.6	16.66	9.54	8.47	8.33	8.29	9.01	10.2	10.38	11.05

TABLE 1. Performance and work profile of $DC_{af,fc}$ SSSP with scale 25 Graph500 input and with varying $send\_threshold$ and $sync\_threshold$ on 64 nodes. The activity counts are sorted in ascending order.

with Graph500 input. With increasing scale, JP suffers from vertex-centric synchronization for large degree vertices. Since *DC* proceeds optimistically when the local per-vertex counter reaches a value of zero, it can avoid waiting time on barrier and is able to finish execution faster.

5.4.5. Performance of SSSP $DC_{af,fc}$ With Various $send_threshold$s and $sync_threshold$s.

Table 1 illustrates how the performance of $DC_{af,fc}$ SSSP varies with different combinations of values for ($send\_threshold$, $sync\_threshold$). Here, the activity counts are sorted in ascending order. This table shows that a right combination of ($send\_threshold$, $sync\_threshold$) parameters is necessary to lower the execution time, even if the activity count is higher. The results are obtained on 64 nodes and with scale 25 Graph500 input. As can be seen from the table, a $send\_count$ value of 10000 and $sync\_count$ value of 2 gives the best performance for $DC_{af,fc}$. In this case, for every 10000 sent messages, we have issued 2 calls with continuation which gives algorithm $DC_{af,fc}$ better opportunity to progress asynchronously. During our experiments, a cursory search for good values for ($send\_threshold$, $sync\_threshold$) parameters resulted in the $(200, 100)$ pair. Thus, we have restricted our search space within the vicinity of 20000 messages and experimented with different combinations of ($send\_threshold$, $sync\_threshold$) for generating 20000 messages. Although the total activity count keep increasing, a right combination of ($send\_threshold$, $sync\_threshold$) value helps to overcome the overhead of executing more work by scheduling work in a timely fashion and gaining better performance in general.

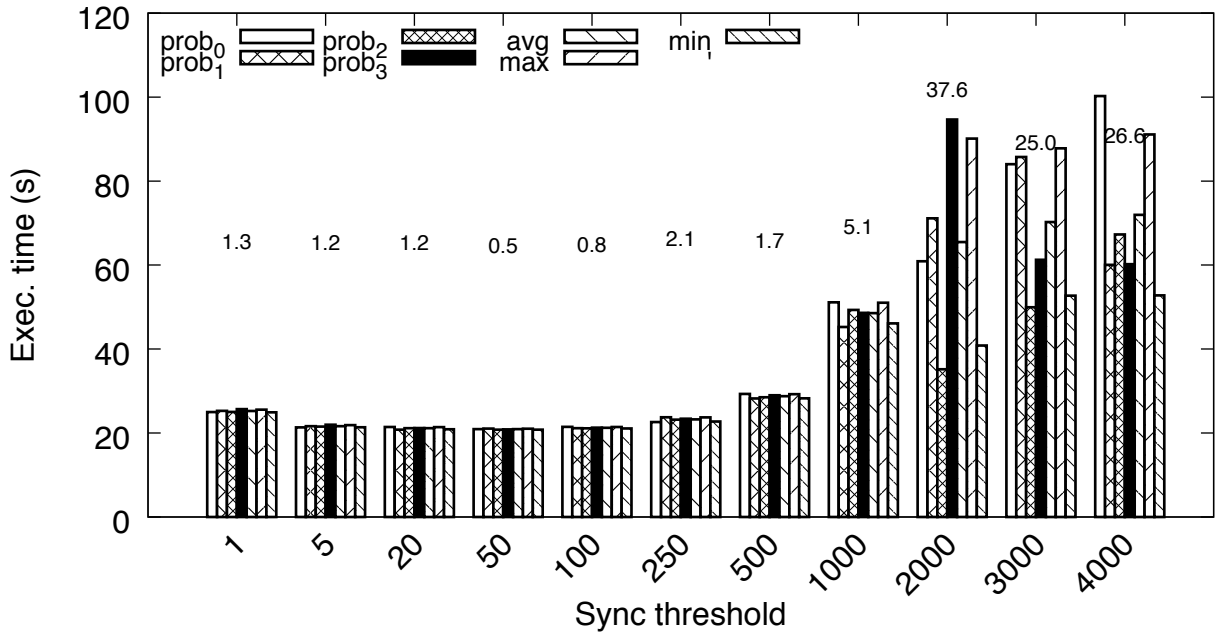


FIGURE 5.9. Execution time and coefficient of variation (CoV) with send threshold 10000 for *DC* SSSP (CoVs shown at the top of clustered bars). We report individual problem execution time, $prob_i$, average time, and adjusted minimum and maximum time from the standard deviation of execution time.

5.4.6. Finding Suitable Runtime Parameter Values. Finding optimal values of the runtime parameters, such as send and sync threshold, by exhaustive search is unfeasible. In designing unordered algorithms, a practitioner needs to make an educated guess about reasonable values of runtime parameters based on available data, which is sometimes hard to collect on a supercomputer due to resource constraints. In this regard, coefficient of variation can be a very useful statistical tool.

When we plot the execution time against different values for a particular runtime parameter (for example, plotting execution time for different sync threshold values at a fixed send threshold of 10000 in Fig. 5.9) we observe that, in the search space of parameter values, there is a range of values that creates a "valley" region. The execution time with parameter values within this valley remains relatively constant between runs (between value 5 and 250 for $sync\_threshold$ in Fig. 5.9). However, outside this valley region, the execution time varies significantly. The coefficient of variation (CoV)((standard deviation/average *100))

of execution time in the valley region is very small. Beyond this "valley" the CoV is very high, which acts as an indicator that we are out of optimal search region. This strategy limits the parameter search space significantly. We observe the similar behavior while choosing a suitable value for sync threshold.

5.4.7. Relation Between Priority Queue Size and Adaptive Frequency in SSSP $DC_{af,fc}$. Adaptive frequency value indicates how many elements to process from the priority queue before transferring control to the runtime scheduler for performing network progression. The smaller the frequency value, the fewer elements are processed from the priority queue, and the algorithm will relinquish control to the runtime scheduler more often. Figure 5.10 depicts how the frequency counter value *freq* in Alg. 12 of $DC_{af,fc}$ for SSSP adapts over time in connection to the priority queue size. For these two quantities, we have seen similar trend across different worker threads. Therefore, we select a representative thread and plot its thread-local priority queue size and adaptive frequency value in Fig. 5.10. At the middle stage of the algorithm execution, $DC_{af,fc}$ receives work items over the network frequently and puts them in the thread-local priority queues. The growing sizes of the priority queues are considered as a manifestation of recurrent successful network activity by $DC_{af,fc}$. At this stage, frequency value is lowered to suggest processing smaller chunk of work items from the priority queue before giving up control to the scheduler to perform network progression. This enables fetching work items from the network more aggressively with the hope of executing less of sub-optimal work. Towards the end of the algorithm execution, fewer work items are sent over the network. To reflect this change, the frequency value starts growing (indicated by arrows in Fig. 5.10). As the frequency value grows, the algorithm processes larger chunk of work items from the priority queue and less often yield to network progression.

5.4.8. Control Release Statistics for *DC* SSSP. As shown in Alg. 12, $DC_{af,fc}$ releases control back to the runtime in 3 situations: the priority queue is empty, the flow-control limit is reached, or the adaptive frequency count of control release is reached. Figure 5.11

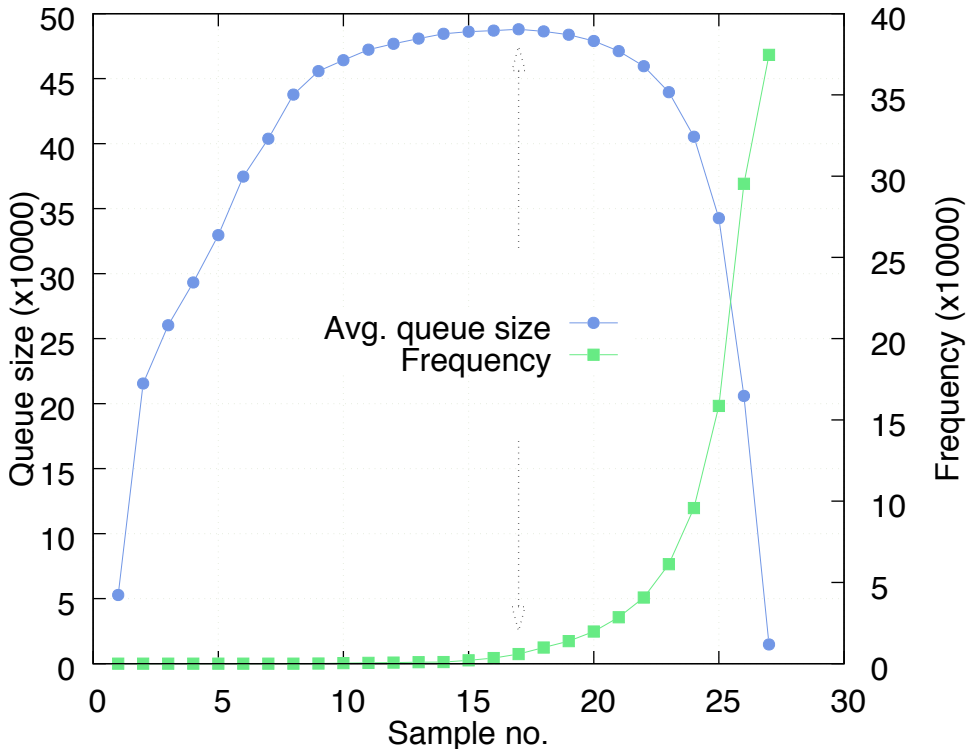


FIGURE 5.10. Variation of priority queue size and adaptive frequency over time in $DC_{af,fc}$ SSSP algorithm with Graph500 scale 27 input and with 8 nodes.

show the yield counts for empty queue, flow-control limit, and the adaptive frequency for Graph500 weak scaling (Fig. 5.3). We conducted several experiments at every scale to find the parameters that result in the best performance. The figure show the statistics for the best performing run (minimum time), median run (median time), and the worst run (maximum time). Empty queue yields are a good performance predictor for Graph500 weak scaling. Flow control count remains relatively stable for the best Graph500 results until 8 nodes, and then it increases for 16 and 32 nodes. With increase in communication, flow control becomes more important. Finally, the adaptive frequency yield count falls with scale since as more work moves over the network, more coalescing is applied and work is delivered in fewer but larger bursts.

5.4.9. $DC_{af,fc}$ SSSP Activity Count vs Execution Time. We show how execution time varies with average activity count for $DC_{af,fc}$ in Fig. 5.12. For reference, we also add a

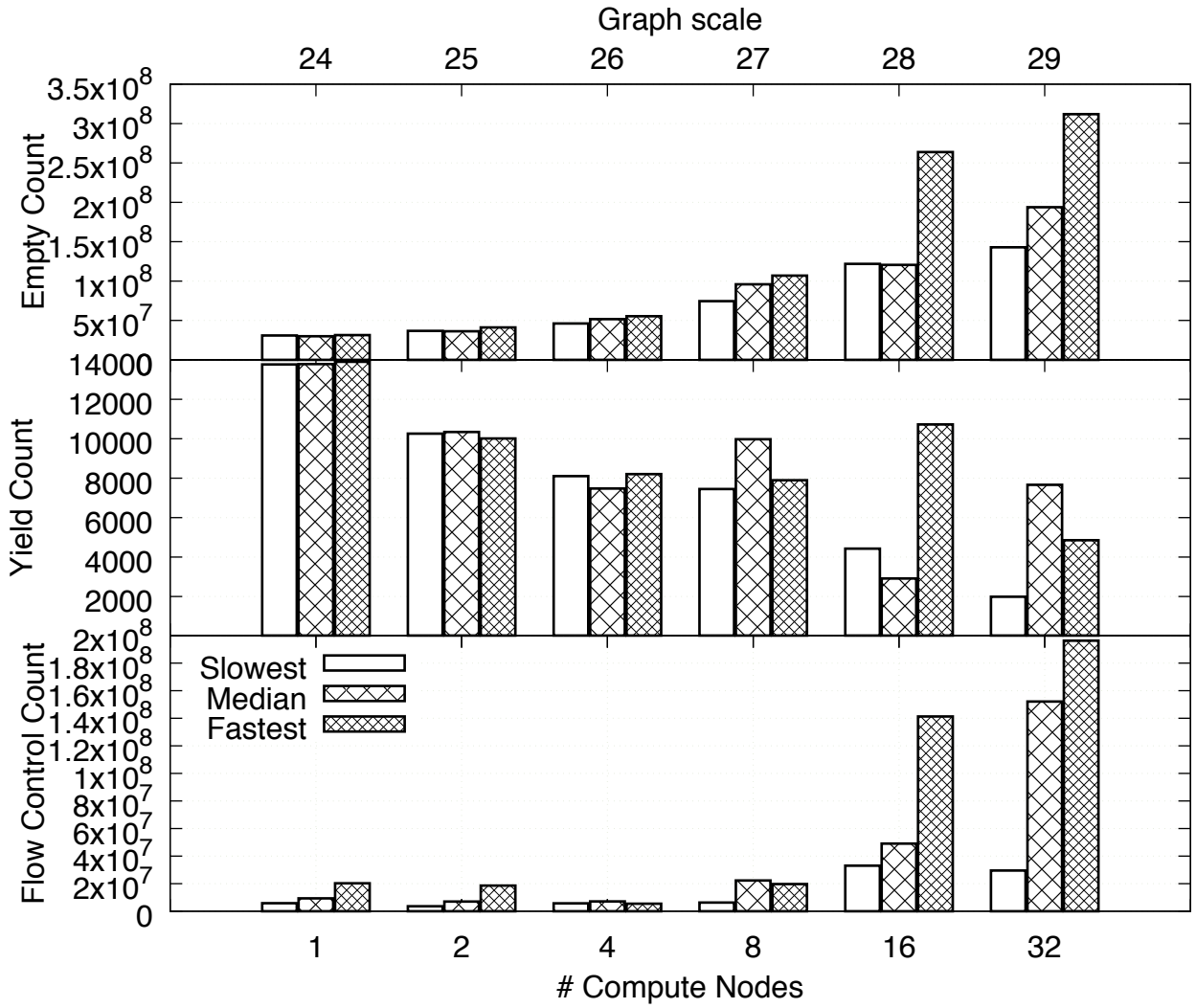


FIGURE 5.11. Statistics of different yield counts for weak scaling results for $DC_{af,fc}$ SSSP algorithm with Graph500 input

line in the plot showing the average total activity count with the *optimal scheduler* for SSSP. This is obtained by setting $\Delta = 1$ and simulating Dijkstra's algorithm. This optimal scheduler performs no label correction, thus optimal in terms of amount of work being executed. However the execution time with the optimal scheduler was 1136 s. $DC_{af,fc}$ achieve best execution time at sample 3, even though the total amount of work done on average is higher than the previous two runs. Each sample in the plot designates a particular $send\_threshold$ and $sync\_threshold$ combination. Beyond sample 3, the general trend is that as the activity count keeps increasing, so does the execution time.

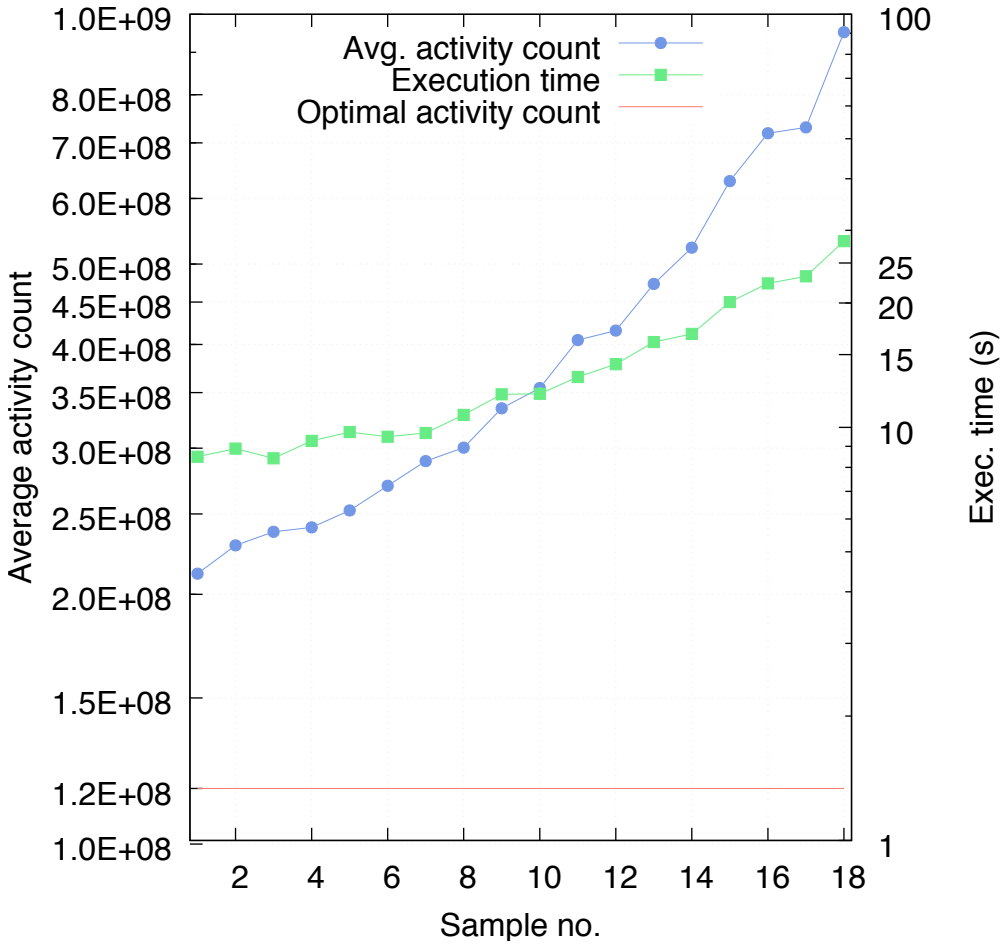


FIGURE 5.12. Relation between activity count and execution time of $DC_{af,fc}$ SSSP with full USA road network with 16 compute nodes

5.4.10. Discussion. We demonstrated that two application-level scheduling techniques, flow control and adaptive frequency of network progress, facilitate better performance of unordered distributed graph computation. $DC_{af,fc}$ algorithms execute more work due to speculative execution of sub-optimal work but perform better than the baseline algorithms. This is due to the fact that with the flow control and the adaptive frequency techniques, instead of waiting on barriers, $DC_{af,fc}$ can schedule work items efficiently by interleaving runtime progress with application work. We observe that input graph structure also has a significant effect on the performance of unordered algorithms such as *DC*. Graph inputs with regular degrees (for example: USA road network) do not contain any skewness or imbalance, hence the opportunity for optimistic execution is insufficient. In such cases,

DC performs comparably as Δ-stepping on few compute nodes. However, DC scales well while the performance of Δ-stepping deteriorates since adding more nodes necessitates more synchronization. Highly skewed graph inputs with power-law degree distribution (such as Graph500) have larger imbalance in the structure, providing *DC* enough opportunity to explore ahead. With such inputs, *DC* demonstrates better weak-scalability.

5.5. Related Work

To the best of our knowledge, Nguyen and Pingali [92], based on the Galois shared memory runtime, first show that performance of algorithms for various irregular applications can improve significantly by selecting right scheduling policies. They evaluated different synthesized schedulers for shared memory systems. Our work explores this concept in distributed settings for graph algorithms.

The concept of plug-in scheduler is not new. Several runtimes [12, 21, 19] provide complete abstraction so as to allow the programmer to design the scheduler from the ground up. However, the interaction between customizable schedulers and communication-bound algorithms is quite unexplored. We investigate the implication of scheduling in terms of communication-bound unordered algorithms, and strive to find a balanced policy to ensure proper mixing of network progress, runtime progress and application level progress.

Also, existing plug-in scheduling policies in several runtimes, for example in OmpSs [21] and StarPU [12], requires the ordering of task execution by assuming a pre-built task-dependency graph. For graph applications, dependency is not know a priori, as it is data-driven. Both of these runtimes assume that a task-dependency graph can be built before starting execution of an algorithm. For example: OmpSs has BFS, work-first, socket-aware , bottom-level aware and affinity aware scheduling. In StarPU a set of common queue designs(stack, FIFO, priority FIFO, deques) and different queuing topologies(central, per-worker) along with the provision for declaring prioritized task is available as different

options for customized scheduling. Also both of the runtimes, in their customized scheduling policies do not concern with how network probing and progress interact with the scheduling.

General and more heavy-weight load balancing and data partitioning techniques are related to our work. For example, Charm++ [67] allows users to dynamically select their preferred load balancing implementation, as well as develop their own using a built-in load balancing API. Legion [14] is even designed around the assumption that the app developers will provide their own mapper implementation to map application to specific architecture. However, our plug-in scheduler is a much more targeted and fine-grained technique that becomes an integral part of the algorithm expression, providing semantic knowledge about best order of execution.

Distributed runtimes sometimes allow programmers to specify task priorities. Xkappi [19], for example, provides push, pop, steal, and activate operation as an interface to the worker queue. These interfaces are limited only to how work is prioritized and retrieved. There is no notion of network flow and probing frequency control mechanism that is directly exposed to the programmer. Charm++ [67] has provision for controlling delivery of messages by allowing users to adjust delivery order of messages by setting the queuing strategy (FIFO, LIFO) as well as two mechanisms for setting priorities (integer and bitvector) [3]. But there is no message throttling mechanism exposed to the programmer. Another recent runtime, Grappa [90] maintains 4 queues: ready worker queue, deadline task queue, private task queue and public task queue for tasks. The deadline task queue manages high priority system tasks. Grappa scheduler allows threads to yield to tolerate communication latency and also has provision for distributed work stealing. Although it has been mentioned in [90] that programmers can direct scheduling explicitly, it is not clear how this can be done from the application level. Lastly, UPC [45] provides topology-aware hierarchical work stealing based scheduling mechanism. Nonetheless, scheduling policies in all these runtimes mentioned above have not been studied in the context of graph algorithms.

5.6. Conclusion

Unordered graph algorithms can be runtime-sensitive. In this paper, we describe how application-level scheduling policies incorporated as runtime plug-in scheduler improve the execution efficiency of the unordered Distributed Control algorithms that eliminate the barrier synchronization and support optimistic parallelism based on asynchronous messaging for work items delivery to workers. Without effective scheduling, the performance of the DC implementation would not be competitive with the baseline algorithms due to undesirable scheduling task flow. However, DC implementation can achieve superior performance with configurable scheduling policies.

CHAPTER 6

Adaptive Runtime Features For Distributed Graph Algorithms

6.1. Introduction

Vastly different approaches [76, 55, 82, 64] for designing distributed graph algorithms mandates versatile support from the underlying runtime system. Specifying fixed policies at execution time for different runtime features can obscure certain *pressure-points* in the runtime. Depending on the executed graph algorithm, with proper *adaptation* of the runtime features, most or all such pressure-points can be adjusted on-the-fly to positively impact the performance of graph algorithms.

In general, programming models [84] for vertex-centric graph algorithms can be classified into two broad categories: Bulk-Synchronous parallel (BSP) model [82](and its variants [58, 106, 37]) and asynchronous programming model [113, 114]. Each of these programming models has different workload characteristics and may require different dynamic support from the runtime. Well-known BSP approaches such as Gather-Apply-Scatter (GAS) [55] model divides the execution into supersteps. Barriers are imposed to prevent any computation strain from digressing too far from the optimal result. It is relatively easier to write, debug, reason, and derive the complexity of algorithms written in frameworks based on GAS model. However, GAS model and its variants thereof suffer from synchronization overhead due to the straggler effect as well as from the bottleneck of distributed lock acquisition on large-degree vertices [84]. Asynchronous graph execution models [113, 114] have recently gained attention of the community due to their potential to unveil more parallelism, suitable for running on top of asynchronous many-task runtimes (AMTs). Such runtimes enable efficient support for lightweight threads, overlapping of communication and computation and scheduling. However, since speculative execution [71] of tasks is a key aspect of asynchronous graph execution model, they can

execute sub-optimal work that may require frequent label-correction as the algorithm progresses. If not carefully designed, performance of such algorithms can deteriorate due to the execution of large amount of sub-optimal work. Expressing a graph algorithm with maximum asynchrony can expose task-granularity at a very fine level.

The remote, irregular memory access pattern, context-switching overhead and high communication to computation ratio in graph algorithms require the underlying runtime to match the appropriate granularity of work execution. This will ensure striking a balance between quantity of work vs. quality of work. Matching a graph expression with proper granularity at the runtime level requires a complementary execution model that can assist in making decision about trade-offs between latency and bandwidth, switching between application-level and runtime-level tasks etc. Static runtime optimizations and coarsening techniques such as message coalescing (aggregation), routing can help in utilizing the bandwidth of the network at the cost of higher latency [108, 86, 99]. Nonetheless, such decision-making first requires identifying a set of "pressure-points" in the runtime and then dynamically adjust certain runtime features on the fly. Encoding such *adaptivity* can speedup the execution time of different graph algorithms. Instead of having a static policy, dynamic execution policy for different runtime features can be adapted over time for an algorithm. In this work, we assume a stateless adaptation: we adjust the parameters based on local knowledge at a particular execution point.

In particular, we investigate how dynamically adjusting message aggregation granularity can speedup certain class of graph algorithms. We also investigate flow control: deciding when to switch to executing application-level work rather than trying to send remote messages in the runtime-level.

In this work, we make the following contributions:

- We identify a set of pressure-points in the lower stack of graph applications aka runtime. Based on graph algorithms' characteristics, we demonstrate how adapting dynamic policies to adjust these pressure-points can benefit such graph algorithms.

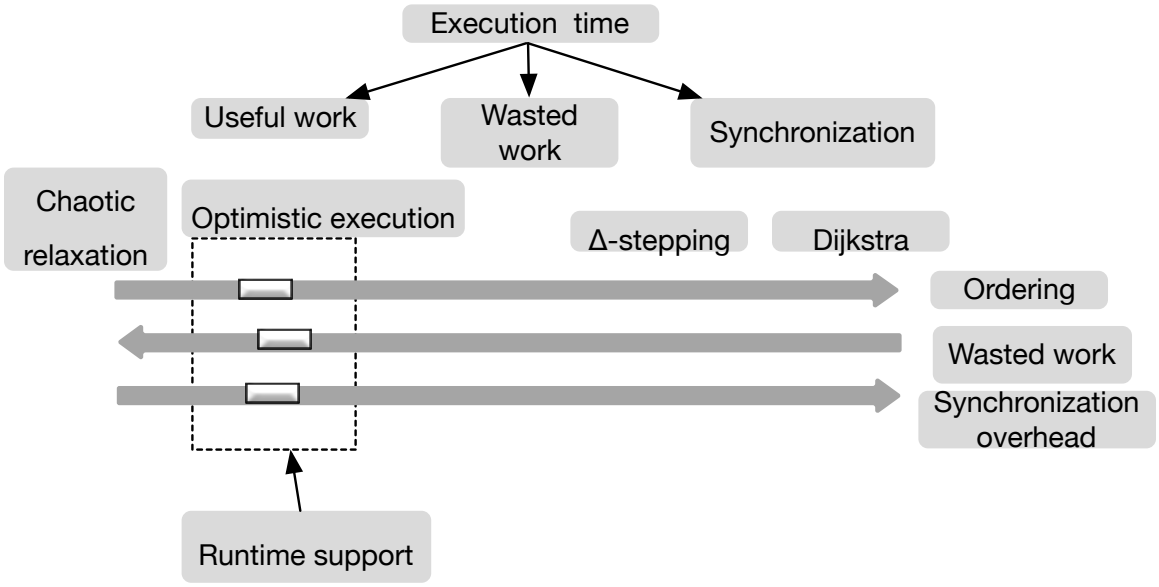


FIGURE 6.1. Finding the sweet spot with adaptivity.

- We demonstrate that adapting dynamic message aggregation policy can speedup graph algorithms that execute in a (relaxed) level-synchronous fashion (for example, Δ-stepping [85] and K-level asynchronous (KLA) [60] single-source shortest paths algorithms).
- We show how dynamically adapting runtime-level flow control mechanism can boost performance of asynchronous graph algorithms that are based on optimistic parallelization.

6.2. Characteristics of Different Classes of Graph Algorithms

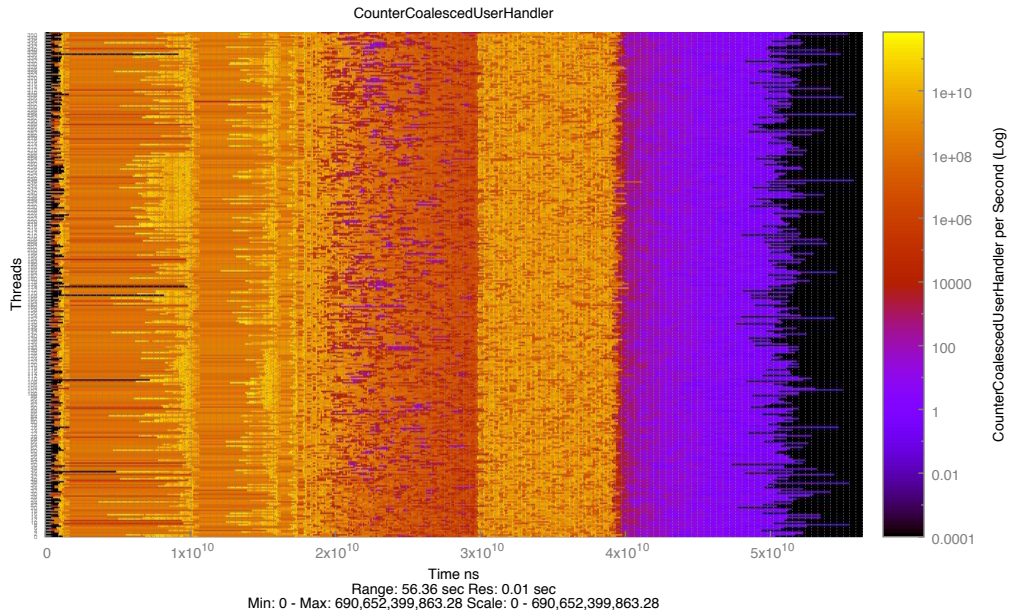
Figure 6.1 shows how different algorithms choose to give preference among work optimality, synchronization, and ordering. The fraction of execution time allocated to address each of these aspects varies across different algorithms. From this perspective, the execution time of a graph algorithm can be broken down into three parts: time to execute useful work, time to execute wasted work and time to synchronize. The following discussion is based on SSSP algorithms in different paradigms, however, in general, other graph applications also have similar characteristics.

Ideally algorithms should only execute useful work. This would require strict ordering of tasks. Dijkstra's algorithm for SSSP [40], for example, is work-optimal. Here work is executed in a strict order of priority. However, the algorithm encounters highest amount of synchronization overhead due to the execution of tasks in the order of priority from the global priority queue. Dijkstra's algorithm is located on one extremity of the spectrum where work-optimality is tied with maximum synchrony. The algorithm is not suitable for distributed execution.

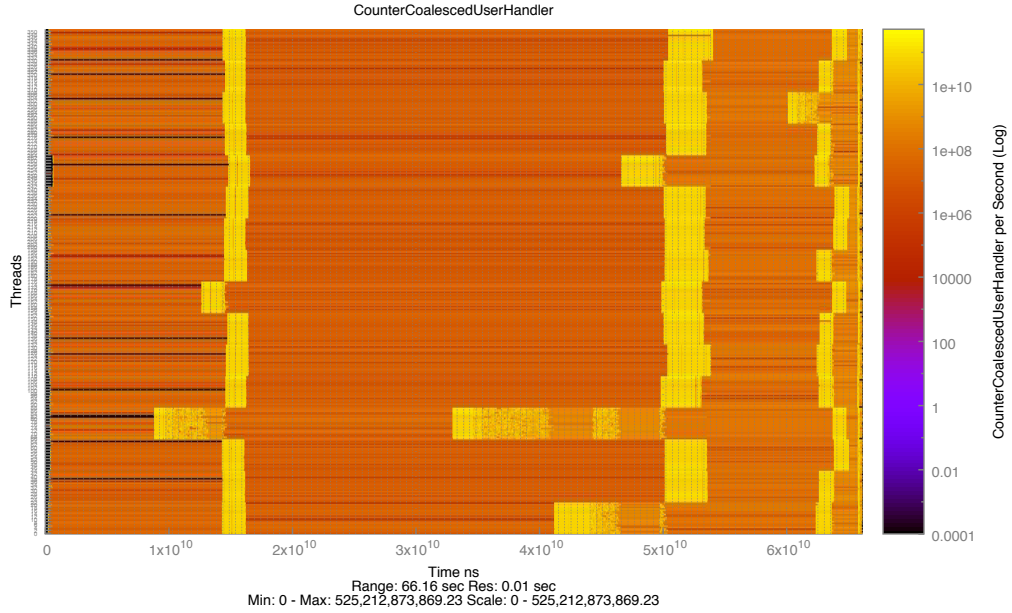
In the other extremity of the spectrum, chaotic relaxation algorithm [28] works by eliminating global barriers and ordering altogether. By eliminating barriers, such execution can proceed without any order and synchronization overhead but can suffer from work explosion by triggering sub-optimal updates to the neighbors. Since tasks are executed as they arrive, intermediate steps of such algorithm do not distinguish better tasks from the worse one and can result in work explosion. Hence, although chaotic relaxation can expose highest amount of parallelism (only syhnchronization required are the atomic updates to the property values of vertices), such execution model wastes resources without any performance benefit.

Δ-Stepping and KLA algorithms (Alg. 14) relax the ordering constraint to allow computation on multiple active vertices within a superstep (bucket) to proceed in parallel. There is no ordering of vertices within a bucket. Splitting the whole execution into a set of supersteps and imposing barriers between two supersteps assist these algorithms to reduce sub-optimal work execution while retaining reasonable amount of parallelism. Nonetheless, such algorithms can suffer from straggler effect where the whole system can not proceed to the next superstep due to a straggler (Fig. 6.2).

Optimistic execution such as Distributed Control or *DC* for short (Alg. 15) tries to find a sweet-spot in between the two extremities discussed above by eliminating global barriers and relying only on local thread-level ordering to avoid work explosion. However, such execution model needs proper runtime support for timely delivery of work from lower-level software stack to the thread-local priority queues.



(A) Δ-Stepping SSSP



(B) KLA SSSP

FIGURE 6.2. Heatmaps of task execution profile (rate) of different SSSP algorithms. The fluctuating task execution rates in Δ-stepping and KLA SSSP algorithm are evident from the uneven stripes of work distribution pattern due to the straggler effects from synchronization. Moreover, at the end of each superstep, the task execution rate gets slower.

Alg. 14: Parallel active-message based relaxed-synchronous algorithms.

In : Graph $\mathcal{G} = \langle V, E \rangle$, levelBound l,

$\forall\, v \in V$: $owner[v]$ = rank that owns v

Out: $\forall\, v \in V$: $property[v]$ = property value of v

```
1  B ← nextB ← empty bucket;
2  level ← 0;
3  property[v] ← init, ∀v ∈ V;
4  enqueue roots → B;
5  message handler explore(Vertex v)
6      enqueue v → B;
7  while B not empty do
8      active message epoch
9          parallel foreach v ∈ B do
10             if Update property[v] then
11                 parallel foreach neighbor w of v do
12                     send explore(w) to owner(w);
13     B ← nextB ← level + l;
```

DC, Δ-stepping, and KLA are label-correcting algorithms (except when $k = 1$) whereas Dijkstra's algorithm is label-setting. The vertex properties (such as distance, component no etc.) calculated by a label-setting algorithm are final and don't change over the course of algorithm execution. On the other hand, label-correcting algorithms can calculate sub-optimal results in intermediate steps and refine these results as the algorithms progress. Hence these algorithms require more careful consideration of runtime scheduling and messaging to strike a balance between asynchrony and ordering.

Alg. 15: Parallel active-message based asynchronous algorithms.

In : Graph $\mathcal{G} = \langle V, E \rangle$,

$\forall\, v \in V$: $owner[v]$ = rank that owns v

Out: $\forall\, v \in V$: $property[v]$ = property value of v

```
1  Q ← empty thread-local priority queues;
2  property[v] ← init, ∀v ∈ V;
3  enqueue roots → Q;
4  message handler explore(Vertex v)
5      enqueue v → Q;
6  active message epoch
7      while Q not empty do
8          parallel foreach v ∈ Q do
9              if Update property[v] then
10                 parallel foreach neighbor w of v do
11                     send explore(w) to owner(w);
```

6.3. The Case For Adaptive Runtime

6.3.1. Major Components Of A Runtime. Runtimes for distributed graph algorithms generally consist of two important high-level components: transport and scheduler (Fig. 6.3).

The transport layer of a runtime (Fig. 6.3) manages various network queues (for sending and receiving messages), and moves messages over the network with the help of a message passing protocol such as GasNet [5], MPI [87], Rsocket [100] etc. Transport may use different protocols (eager vs rendezvous) for exchanging messages, can support different levels of thread safety (serial, funneled, multiple etc.) for accessing network buffers, and can choose different communication paradigm (two sided, one-sided, collectives) [50]. Hence different combinations of such choices would require keeping proper track of the requests being made.

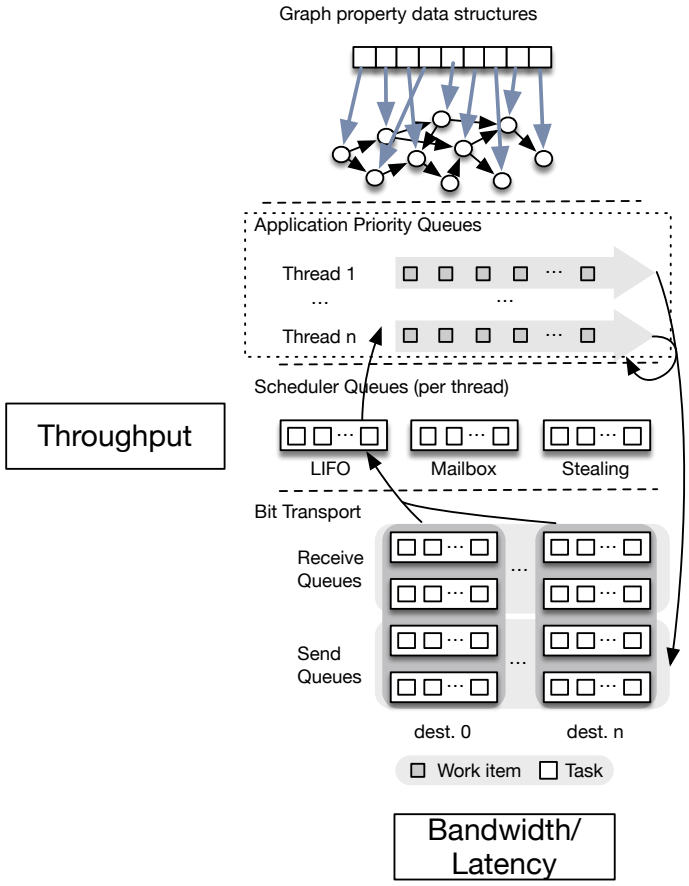


FIGURE 6.3. Overview of the system stack for graph applications.

The scheduler (Fig. 6.3) is responsible for maintaining data structures for task execution (including work stealing, mailboxes for threads) and for choosing tasks and executing them. These tasks include application-level work, runtime book-keeping tasks such as servicing the network, termination detection etc. Ideally it is preferable to match the throughput of the runtime scheduler with the network bandwidth (both in terms of sending and receiving messages). However, local computation and memory accesses are faster than remote updates. Hence graph algorithms are mostly communication-bound (an exception is the algorithms for triangle counting , where the computation to communication ratio is generally higher).

6.3.2. General Runtime Mechanisms For Optimized Graph Algorithms.

6.3.2.1. *Message Coalescing.* As discussed, graph algorithms are communication-bound and generate a large amount of irregular remote memory access requests. Such requests

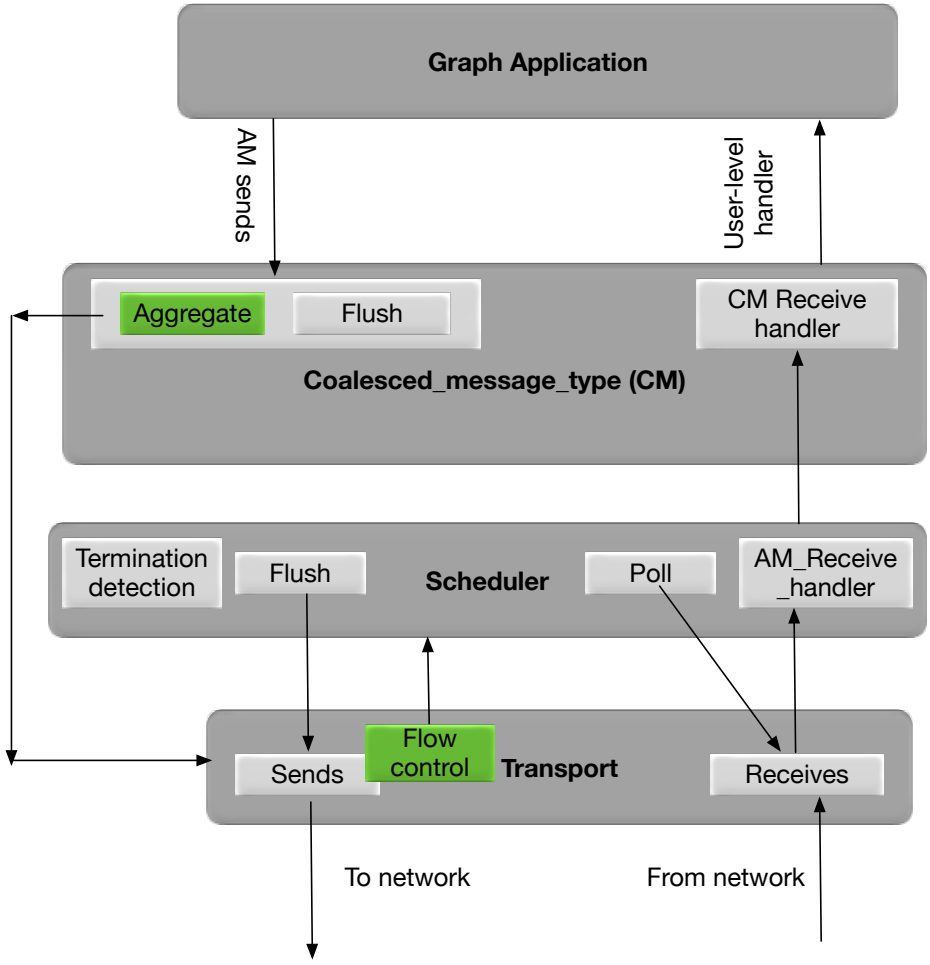
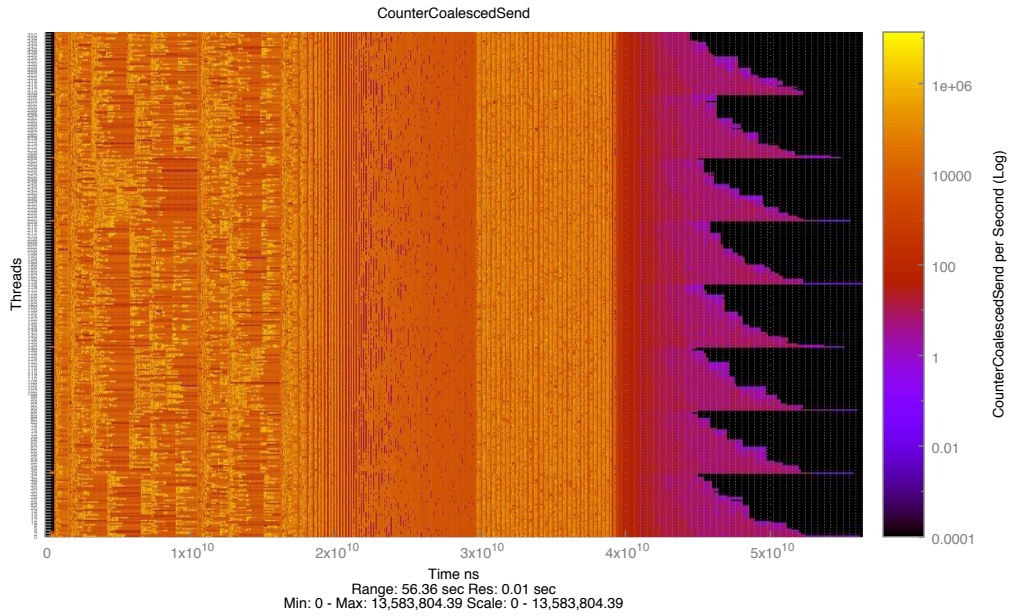
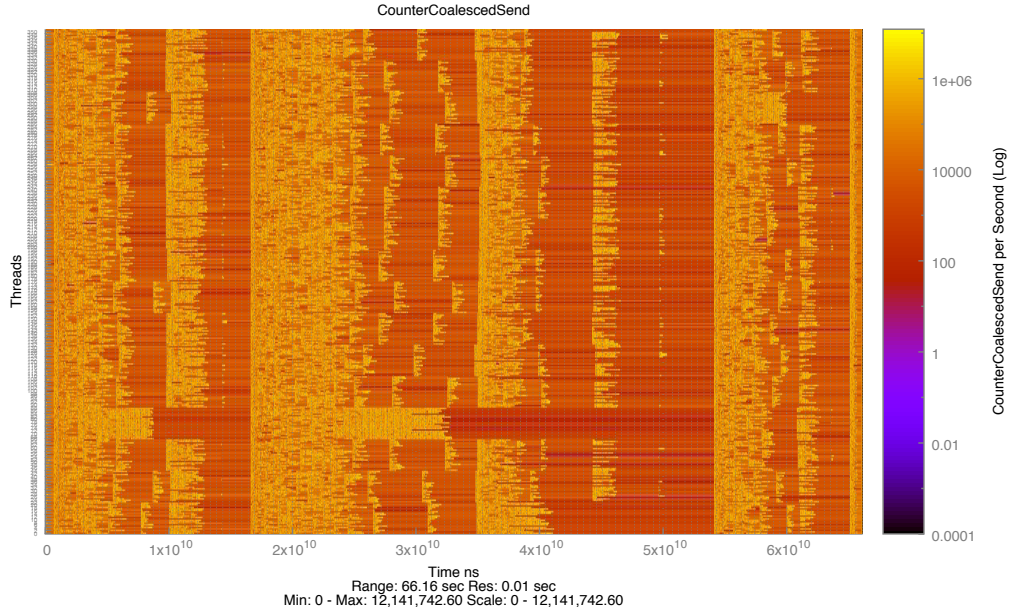


FIGURE 6.4. Candidate mechanisms to accelerate graph applications.

translate into sending huge amount of small active messages targeted for remote ranks. To avoid overheads of sending individual small messages and efficiently utilize the bandwidth of the network, technique such as *message coalescing* (aggregation) (Fig. 6.4) has been used in practice [108, 86, 99]. As shown in Fig. 6.4, in active-message [105] based runtimes, graph applications register messages to be sent over the network as coalesced message type, along with registering a handler (AM_Receive_handler, short for active-message receive handler) that will be invoked transparently by the receiver of such messages. When a message is generated by a graph algorithm to be executed at a remote locality (such as a vertex-distance update message), it is handed over to the transport. The transport appends the message in the appropriate destination's coalescing buffer. If the buffer is full according to some pre-specified parameter value (in terms of no. of bytes, no. of



(A) Δ-Stepping SSSP



(B) KLA SSSP

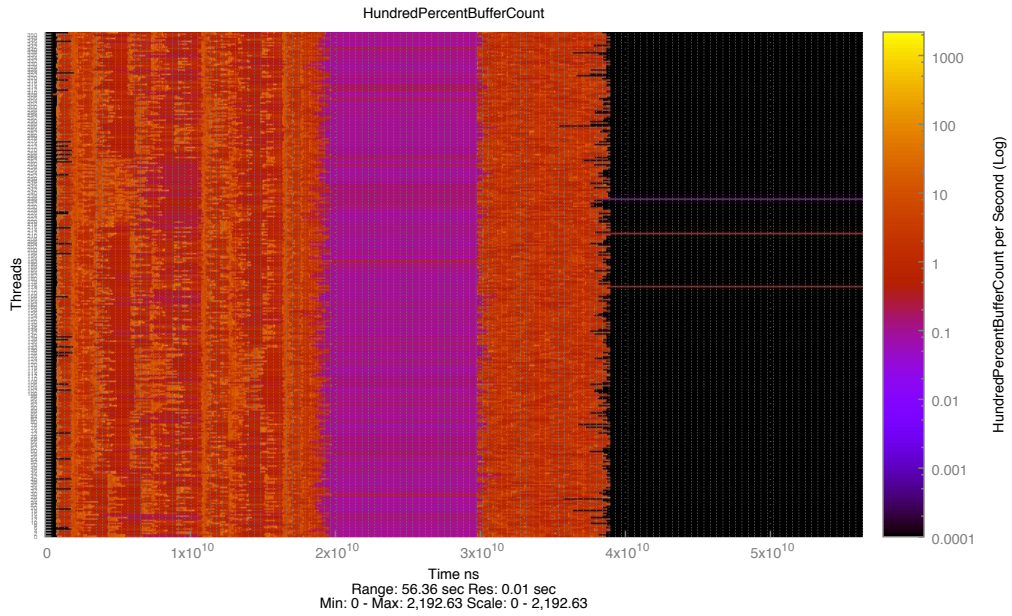
FIGURE 6.5. Heatmaps of message send rate of relaxed-synchronous SSSP algorithms.

messages etc.), the buffer is handed over to the underlying messaging layer such as MPI for transportation. Coalescing messages utilizes bandwidth of the network more efficiently at the expense of latency.

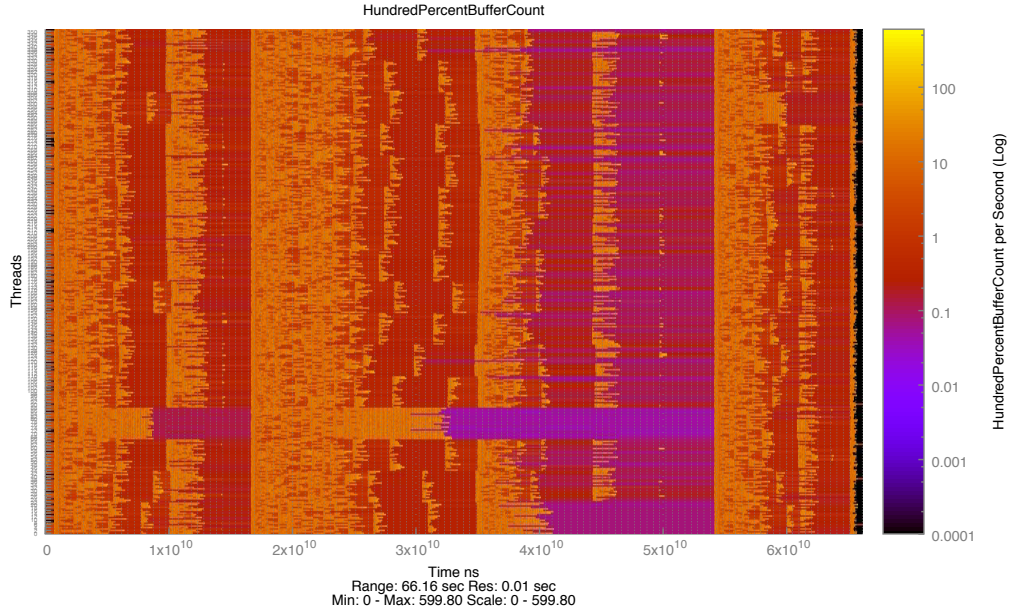
However, such fixed parameter value for coalescing may restrain certain graph algorithms from getting better performance. For example, many relaxed-synchronous graph algorithms such as Δ-stepping and KLA algorithms (Sec. 6.2), however, do not sustain constant workload within each of the supersteps. As shown in Fig. 6.5, when the algorithms draw near to the end of supersteps, message sending rate becomes slower. With a fixed coalescing message buffer size, a runtime would require to either wait to fill in the coalescing buffer or need a timer to trigger sending the (partially) coalescing buffers. Waiting for such triggers can result in loosing performance for algorithms with relaxed synchrony.

6.3.2.2. *Runtime Flow Control. Flow control* mechanism in the transport layer of the runtime decides whether to progress the network for some time or transfer control to the runtime scheduler to execute other tasks (Fig. 6.4). The purpose of flow control is to ensure that there is a right balance between progressing the network and performing algorithmic level work. Absence of flow control can become problematic if receivers fail to keep up with processing received messages from different senders due to eager sends.

Fixed policy for flow control can limit the performance of asynchronous graph algorithms. For example, if the parameter value for flow control is set to 50 to indicate that if, at a certain point of execution, there are more than 50 outstanding MPI requests, then the control should be transferred to the runtime scheduler, it may be inflexible at certain stage of an algorithm execution. If a receiver side faces bursty communication from several senders simultaneously, it may be advantageous to throttle the back-pressure from the sender side. Asynchronous graph algorithms try to strike a balance between ordering and suboptimal work. If proper consideration is not given, sub-optimal work execution can generate work exponentially, thus receivers can get overwhelmed with messages without making significant progress in request tracking in the transport.

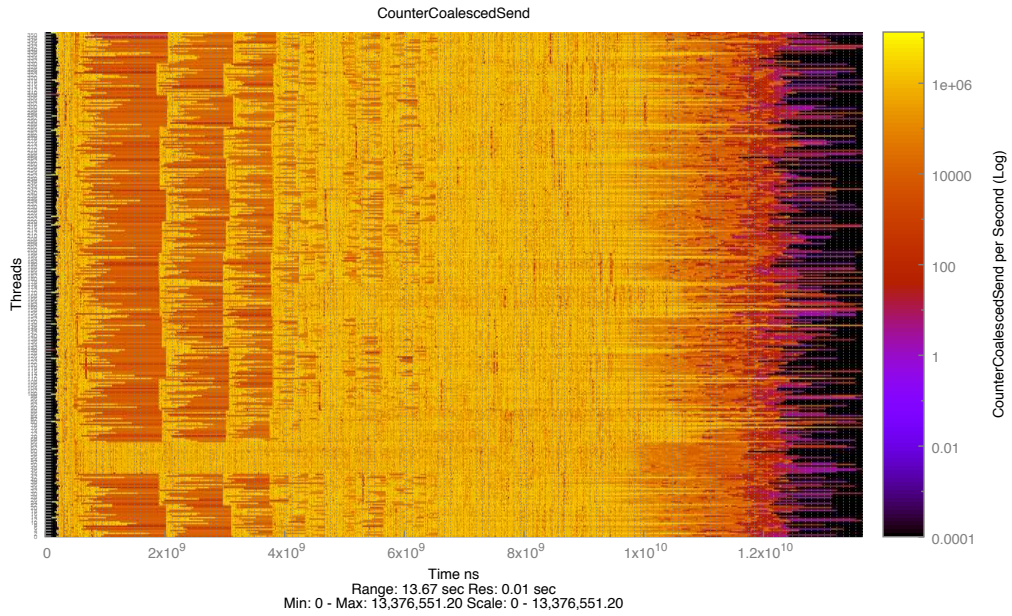


(A) Δ-Stepping SSSP

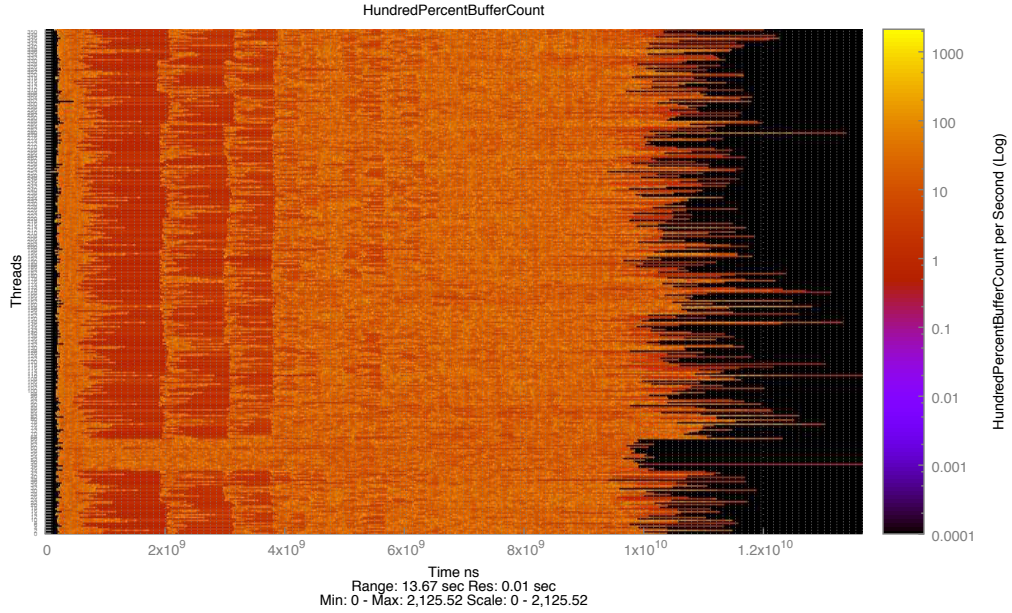


(B) KLA SSSP

FIGURE 6.6. Heatmaps of sending full message buffers of relaxed-synchronous SSSP algorithms.



(A) Message send rate



(B) Full message buffers

FIGURE 6.7. Distributed Control message sending profile.

Static Policy. With static message coalescing, at the beginning of an algorithm, a fixed coalescing buffer size is specified for the runtime. Based on the provided fixed size, a coalescing buffer is created for each destination by the runtime on each compute node. Buffers are shared by all worker threads in the system. However, to avoid contention, these buffers are implemented as circular buffers such that messages are written at one end of the buffer while they are read from the other end, with the sizes adjusted accordingly with atomic operations. When the application layer hands over a message to the coalesced-message-type layer for sending, a check is first done by the runtime to see whether the buffer of the target destination is full. In such case, full buffers are sent over the network first. Otherwise, the new message is queued in the appropriate destination coalescing buffer.

Flushing heuristic. Sometimes, to avoid the delay of delivering messages, runtime executes flush task to send partially filled buffers to the destinations. To flush messages, the runtime checks whether the coalescing buffer size for a particular destination has increased since last visit. If the buffer size has not grown since last visit, it is an indication that the current rank is not generating enough messages for the particular destination. So instead of waiting to accumulate more messages, the runtime sends partial buffer to that destination.

Adaptive policy. Generally, algorithms that are designed to execute in asynchronous fashion are expected to generate messages at higher rate. With proper runtime support, these algorithms can sustain nearly constant message volume at the middle stage of algorithm execution (Fig. 6.7). Thereby the runtime will send full buffers to remote destinations during most of the execution time. In contrast, relaxed-synchronous algorithms such as Δ-stepping and KLA algorithms, generate messages in much lower rate at the end of each superstep (level) execution (Figs. 6.5 and 6.6). Waiting for a time-out or the flushing heuristic to kick in to flush the buffers can be detrimental to the performance of these

algorithms. Such waiting periods can accumulate over several supersteps and become significant.

For such relaxed-synchronous algorithms, where message generation rate is not sustainable, adaptive policy can be useful. In our adaptive message aggregation policy, in regular interval, when an active message is sent from the application layer to coalesced message type layer, we check whether the message generation velocity has fallen behind a certain threshold. Depleting message generation velocity is an indicator that current superstep of the algorithm is reaching to an end and the runtime should send messages more aggressively, rather than waiting for a time-out or the buffers to get filled-up. In this scenario, the runtime flushes the partially-filled coalescing buffers.

6.4.1. Adaptive Flushing. In our previously discussed approach for adaptive message aggregation, an assessment to flush messages is performed by the runtime when a flush task is directly executed from the scheduler Fig. 6.4. Additional assessment to flush messages can be done before executing an active message handler from the scheduler. Upon receipt of a coalesced message, a pre-registered active message handler generally spawns a set of user-level handlers Fig. 6.4 and hence can be long-running. Before executing such task from the scheduler FIFO queue, the current worker thread can attempt to obtain a lock on the transport and, if successful, flushes all the destination buffers. Interleaving flush tasks with execution of active message handlers can boost performance of graph algorithms.

6.5. Adaptive Flow Control

At any instance of time, a runtime can execute either runtime-level tasks or application-level workitems. While servicing the network at the runtime-level, consider a situation when the remote destinations can not keep up with receiving messages from the sender. In such case, to regulate back-pressure, an adaptive flow control policy in the runtime

can become handy. In our adaptive flow control implementation, we maintain a flow-control threshold for each destination. When a message is handed over to the lower-level runtime to be sent over the network, a check is performed by the runtime to see whether the number of outstanding messages for that particular destination has crossed the flow control threshold. In such situation, instead of pushing more messages over the network to that particular destination, the control is transferred to the application-layer to progress application-level task. The flow control threshold value is also increased with the anticipation that the destination is currently unable to process received messages faster. However, if the number of outstanding messages is smaller than the current flow control threshold, the threshold is lowered for the next iteration so as to send messages more aggressively to the destination.

6.6. Experimental Results

6.6.1. Experimental Setup.

6.6.1.1. *Runtime.* We have implemented our algorithms in the AM++ runtime [108]. AM++ is based on active messages, where sends are explicit but receives are implicit. AM++ can run programs in two different *epoch* execution models. In *scoped epoch* model, runtime progress is executed once all algorithmic level work are finished for an epoch. In *end-epoch test* model, runtime and algorithmic work are interleaved. Runtime progress involves progressing the network, polling, termination detection etc. All relaxed synchronous algorithms are implemented in the scoped epoch model, since each such epoch can act as a global barrier between supersteps. Distributed control algorithms are implemented in the end-epoch test model. The global quiescence is checked periodically by the AM++ runtime performing global reductions on two counters: active and finished, in two phases and checking whether their difference has reached a value of zero in subsequent phases (similar to the four-counter based algorithm proposed by Sinha et al. [103]).

6.6.1.2. *Dataset.* We evaluate the performance of our algorithms with RMAT graphs [73]. RMAT graph generator works by dividing the adjacency matrix of a graph into 4 separate

quadrants. The probability of an edge between two vertices from different quadrants is specified by 4 parameters: a, b, c, and d. We have experimented with two types of graphs: Erdős-Rényi (RMAT-ER with $a = b = c = d = 0.25$) and Graph500 [88] (with $a = 0.57, b = 0.19, c = 0.19, d = 0.05$). In our plots, a graph of scale x denotes a graph with 2^x vertices. Each vertex in the RMAT graphs has an average degree of 16 (directed), for a total of $2 * 16 * 2^x$ edges, considering undirected edges. In the plots for our scaling experiments, X axes have a one-to-one correspondence and indicate the scale of the input graph and the corresponding number of compute nodes employed in each experiment.

6.6.1.3. *Hardware.* We have conducted our experiments on a Cray XC30 system. Each compute node on the XC30 system consists of two Intel Xeon E5 12-core x86_64 2.3 GHz CPUs with hyper-threading enabled (up to 48 hardware threads per node) and of 64 GB of DDR3 RAM. All XC30 nodes are connected through the Cray Aries interconnect.

6.6.1.4. *Compiler Options.* We have compiled our code with gcc 7.2.0 and with optimization level '-O3'. Additionally, single node experiments were performed with networking turned on.

6.6.1.5. *Graph Representation.* The graph is distributed across different compute nodes using block distribution and is represented with a distributed compressed sparse row (CSR) data structure. For SSSP algorithms, edge weights have been chosen randomly within the range of $[0, 255]$. In all weak scaling plots we have truncated the execution time to 1000s. We report our experimental results as averages of 8 runs for weak scaling results. We evaluated different Δ and k values and have set $\Delta = 3$ and $k = 1$ in these experiments, as these values have shown to result in best-performing algorithms with specified weight range.

6.6.2. Adaptive Coalescing Results: Relaxed-synchronous Algorithms.

6.6.2.1. *Single-source Shortest paths algorithms.* Figures 6.8 and 6.9 show percent improvement of Δ-stepping algorithm with adaptive coalescing policy over static policy discussed in Sec. 6.4 for weak scaling results. In both cases, at larger scale, Δ-stepping algorithm achieves better performance with adaptive coalescing technique. As the number

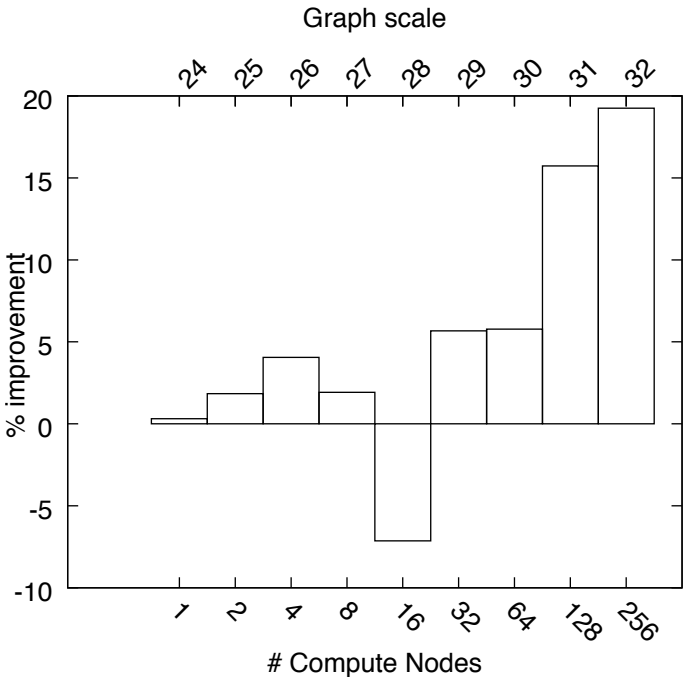


FIGURE 6.8. Percent improvement of execution time in weak scaling with adaptive coalescing for Δ-stepping SSSP with Graph500 input.

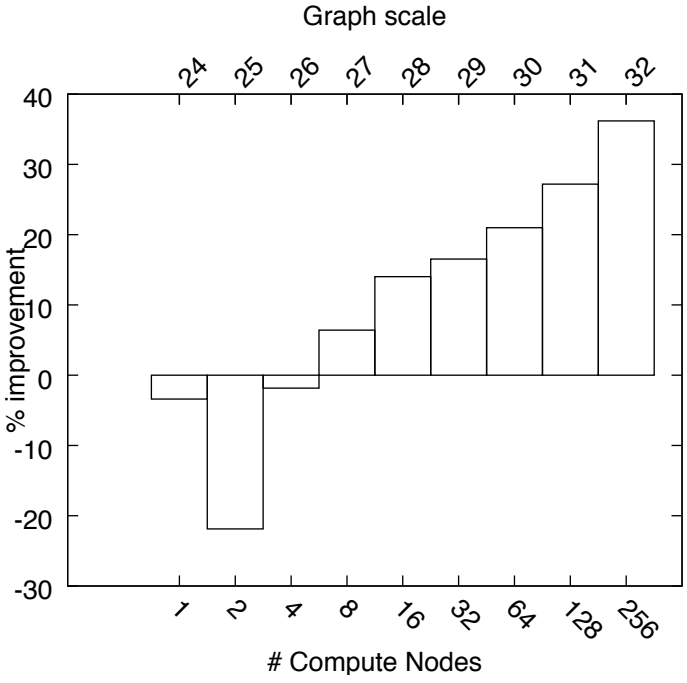


FIGURE 6.9. Percent improvement of execution time in weak scaling with adaptive coalescing for Δ-stepping SSSP with RMAT-ER input.

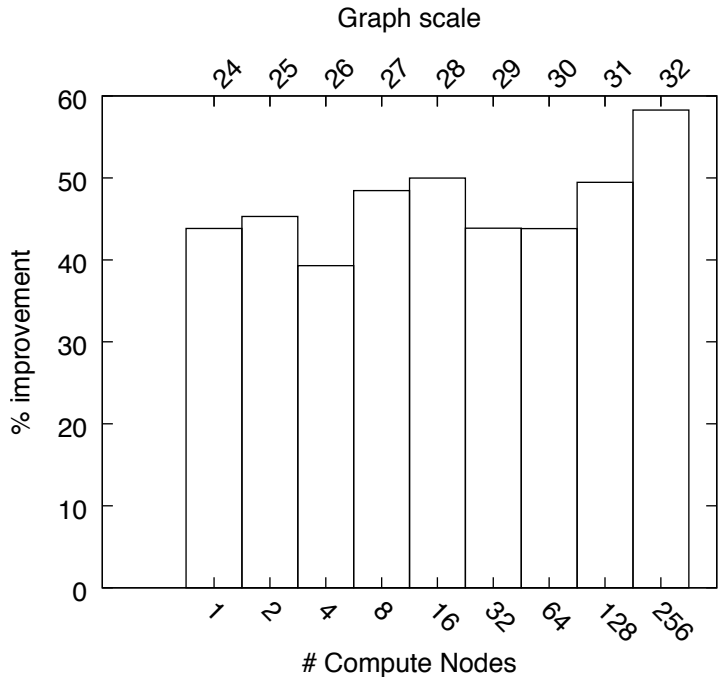


FIGURE 6.10. Percent improvement of execution time in weak scaling with adaptive coalescing for KLA SSSP with RMAT-ER input.

of coalesced destination buffers increases with larger number of compute nodes, message aggregation benefits from dynamic adaptivity by reducing the waiting time to flush the buffers when the algorithm is closer to the end of each superstep.

Additionally, we also report percent improvement of KLA algorithm with adaptive coalescing policy in Fig. 6.10 with RMAT-ER graph. Here, we observe consistent improvement of execution time as we increase the scale of the input along with number of compute nodes. In particular, we see $\sim 60\%$ improvement of execution time with 256 compute nodes. On a single node, we have run our experiments with networking turned on, with 2 ranks per node.

6.6.2.2. *Connected component Algorithm.* We have also applied our dynamic policy for adaptive coalescing to Shiloach-Vishkin (SV) connected component algorithm that is based on two synchronization phases: hooking and shortcutting. With adaptivity, the performance of the algorithm improved up to $\sim 24\%$ (Fig. 6.11).

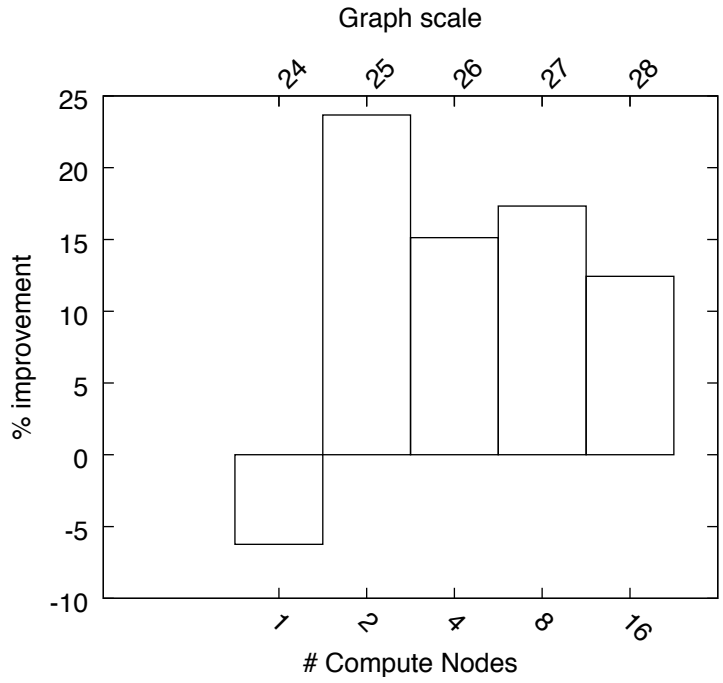


FIGURE 6.11. Percent improvement of execution time in weak scaling with adaptive coalescing for CC SV with Graph500 input.

6.6.3. Adaptive Flow Control Results: Asynchronous Algorithms.

6.6.3.1. *Distributed Control Single-source Shortest paths algorithm.* Figures 6.12 and 6.13 report the improvement of Distributed Control based SSSP algorithms with Graph500 and RMAT-ER inputs. At larger scale, we observe better performance of DC. In particular, with Graph500 input, DC enjoys better improved performance. Graph500 input has skewed degree distribution and contains several high-degree vertices that can cause load imbalance in computation. Adaptivity of flow control helps asynchronous algorithms such as DC to interleave runtime tasks with application work more efficiently. Such interleaving of tasks and work can assist in the elimination of sub-optimal work by prioritizing work and helping the scheduler to progress the network at proper time.

6.6.4. Distributed Control Connected Component Algorithms. We report performance improvement of DC based connected component algorithm with RMAT-ER and Graph500 input in Figures 6.14 and 6.15. At scale, this algorithm sees $\sim 20\% - 30\%$ performance benefit with adaptive flow control policy.

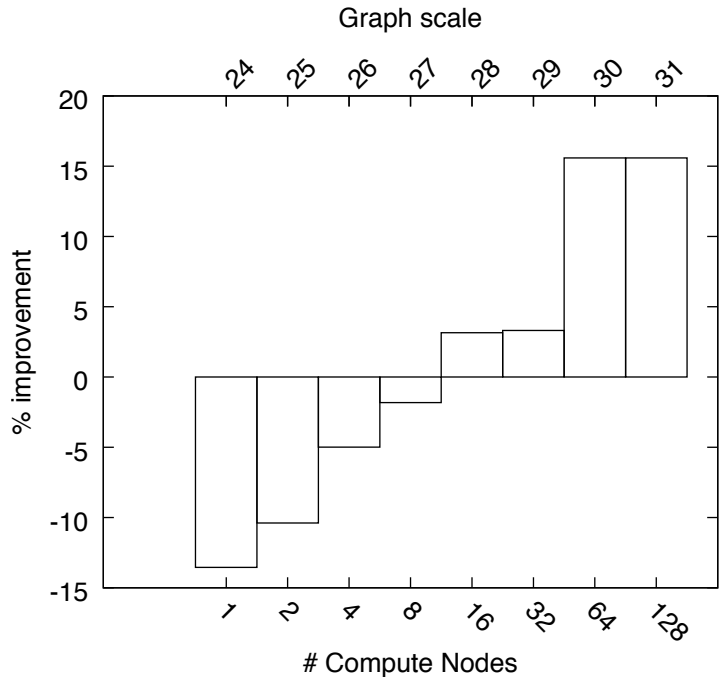


FIGURE 6.12. Percent improvement of execution time in weak scaling with adaptive flow control for DC SSSP with Graph500 input.

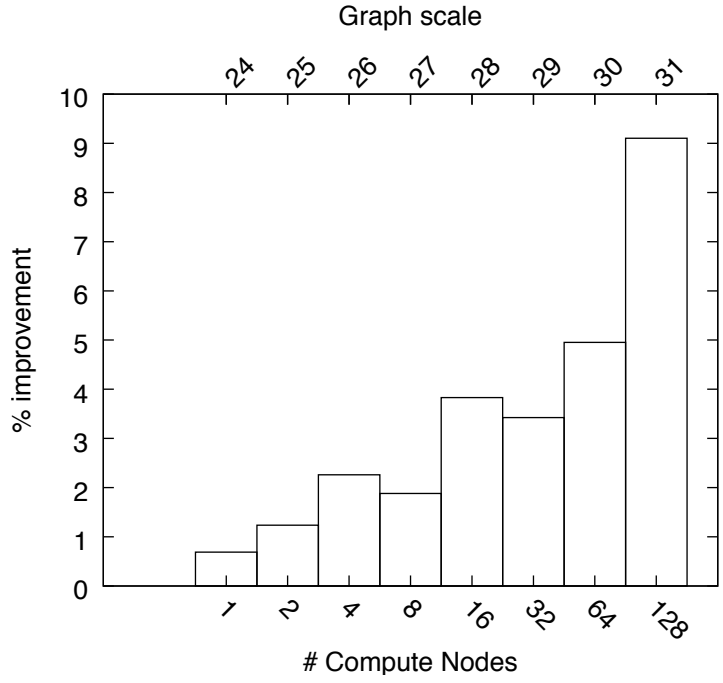


FIGURE 6.13. Percent improvement of execution time in weak scaling with adaptive flow control for DC SSSP with RMAT-ER input.

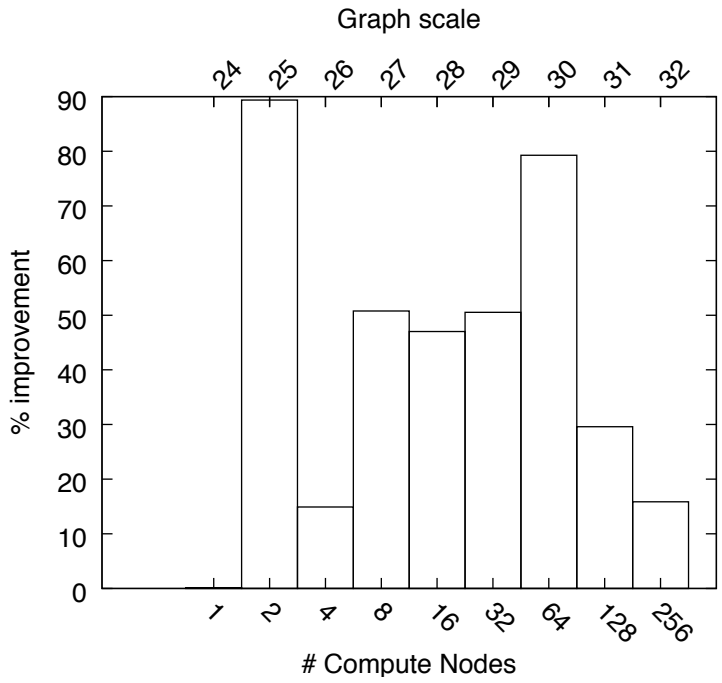


FIGURE 6.14. Percent improvement of execution time in weak scaling with adaptive flow control for CC DC with RMAT-ER input.

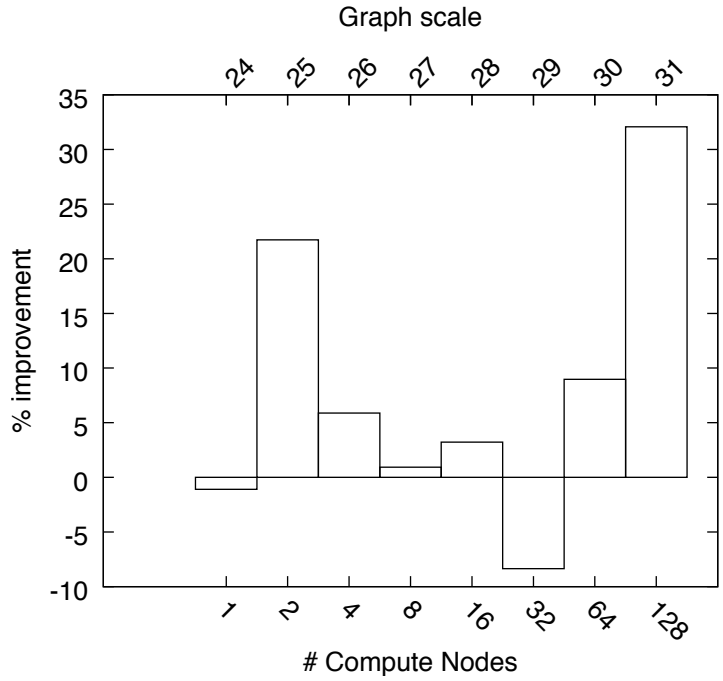


FIGURE 6.15. Percent improvement of execution time in weak scaling with adaptive flow control for CC DC with Graph500 input.

6.6.5. Experiments With Real-world Graphs. We also experimented with two real world graphs: Friendster [74] social network and roadNet-CA [4] road network for assessing the benefit of adaptive message aggregation. Friendster has 65M vertices and 2.9B edges. roadNet-CA has 1.9M edges and 5.5M edges. With Friendster input and adaptive coalescing, Δ-Stepping and *KLA* obtained an improvement of 8.92% and 6.74% respectively. With roadNet-CA input and adaptive coalescing, Δ-Stepping and *KLA* obtained an improvement of 6.5% and 7.73% respectively.

6.7. Related Work

Dynamic message aggregation: In [32], the authors proposed two aggregation strategies for time-warp simulators on unicore systems: Fixed aggregation windows (FAW) and simple adaptive aggregation windows (SAAW). These approaches are based on the age of the aggregate. The time window is adjusted based on the message rate over the execution time of an algorithm. Their approach of adaptivity is based on the temporal aspect of aggregation. However, it has been shown in [97] that since posting time of reception calls has no impact on performance, it is not possible to use an age criterion for message aggregation. In contrast, our approach adjust the spatial aspect of the aggregation, where the buffer sizes are adjusted by the runtime over the execution time of an algorithm.

Admission control Policy. To avoid congestion, Luo et al. [79] identified two types of congestions: rate congestion due to too many concurrent injections and concurrency congestion due to too many cores being active. The authors proposed an application-level admission control mechanism for messages for one-sided communication in UPC to proactively mitigate the adverse effect of congestion. However, this has not been explored in the context of highly asynchronous two-sided communication with coalesced messaging capability.

Active layer extension for MPI. In [33], the authors proposed an extension to MPI to incorporate a set of optimizations for communication: dynamic message aggregation to reduce the send overheads and infrequent polling to reduce the receive overhead of

messages. However, dynamicity of aggregation comes from taking a fixed buffer size for aggregation from the user that depends on the application and use timer for message flushing. Hence, instead of using a fixed buffer size for all the applications, user-provided buffer size can be used. While they envision buffer sizes to grow and shrink during application execution, experimental results have only been reported with different fixed buffer size for LU decomposition and discrete event simulation. We demonstrate the benefit of dynamically adapting aggregation buffer sizes with message velocity during different phases of an algorithm.

Application-level scheduling policies. Previously, in [51], we have demonstrated that plug-in scheduling policies in the *application layer* for asynchronous distributed graph algorithms such as Distributed Control can be beneficial for achieving better performance. Our current work investigates how adaptivity of runtime features in the *runtime* layer can help both relaxed-synchronous and asynchronous graph algorithms.

Graph partitioning and load balancing. Various graph partitioning techniques exist [22] to minimize communication volume by finding a reasonable cut. However, this requires a separate graph preprocessing step to be executed to get a balanced partition before starting an algorithm execution. Hence most graph partitioners are completely algorithm-agnostic. Moreover, for many graphs, no good partitioner exists. Runtimes such as Charm++ [67] provides user with APIs for load balancing. However, the techniques are heavyweight and require the computation to temporarily suspend for balancing before proceeding. Additionally, the efficiency of such load balancers has not been studied in the context of fine-grained graph computation. Our approaches do not depend on any preprocessing step, are very lightweight to execute and take into consideration online workload characteristics of different graph algorithms.

6.8. Conclusion

In this work, we identify a set of candidate runtime features: message coalescing and flow control that can be adjusted during the execution of a graph algorithm. We

differentiate the workload characteristics of two types of graph algorithm programming models: relaxed-synchronous algorithms based on supersteps and asynchronous graph algorithms. We demonstrate that adapting dynamic message coalescing technique can accelerate each superstep that results in overall improved execution time for relaxed-synchronous graph algorithms. Additionally, we show that interleaving the execution of runtime task and algorithm level work by adjusting network progress dynamically in response to a hint of back-pressure from the receiver can improve the execution time of asynchronous graph algorithm.

CHAPTER 7

Future Directions

In this thesis, we have proposed algorithms and runtime design techniques for graph applications that achieve scalability by eliminating the bottlenecks associated with global synchronization and vertex-centric barriers. In this chapter, we discuss possible future directions that can be pursued.

7.1. Minimizing Synchronization In Dynamic Graphs Algorithms

Our current work investigates algorithmic techniques with *static* graph inputs. However, with the proliferation of research activities in the fields of artificial intelligence, knowledge representation and information security, there have been growing interests about *dynamic* graphs. A promising extension to the current work can investigate techniques to minimize synchronizations in graph algorithms for *dynamic* graphs [41]. In a dynamic graph, the graph structure/topology changes over time as edges and vertices are added and deleted. In a distributed setting, the frequency of updates on a dynamic graph poses several challenges in designing scalable algorithms. For example, designing scalable distributed data structures for representing dynamic graphs is vital for efficient execution of a dynamic graph algorithm. Some customized data structures for efficient dynamic graph algorithms on shared memory systems have been proposed [41, 57]. In a distributed setting, maintaining a dynamic graph is more challenging. A recent work [49] has proposed a data structure, DISTINGER, for processing streaming data in distributed setting. However, due to its design based on the master-slave configuration for updates, DISTINGER suffers from the bottleneck of having single-point of entry (master) for updates. In contrast, active message based approaches such as ours are well-positioned to propagate update to the owner, without re-routing the updates through the master.

Depending on the set of requirements on the updates/modifications of the graph data structures, different models for dynamic graphs have been proposed: batch [41], streaming [13], semi-streaming [48]. These models consider trade-offs among various metrics such as the timeliness of an answer to a query, memory usage, error bounds (approximate) on an acceptable answers etc. For example, in the streaming model of graph computation [13], a sequence of edges with arbitrary permutation is presented to the algorithm, one at a time. In semi-streaming model, an algorithm is executed following some constraints [48]: the amount of space for processing the graph is fixed beforehand, the number of passes allowed over the stream of edges is also fixed and the time to process each edge is also limited.

Future investigation of dynamic graphs can be carried out from at least two perspectives:

- Designing efficient data structures for dynamic graphs, that would
  - require minimal synchronization,
  - utilize local fine-grained locking, and
  - reduce contention on the data structures

  across different compute nodes in the distributed setting, as the updates come in.
- Identifying dynamic graph kernels and classify them into different groups, if possible.
- Designing algorithms for dynamic graphs that avoid/minimize synchronization and vertex-centric barriers.

We envision that our label-correcting approach will be particularly suitable for dynamic graphs, with minimal need for synchronization across different nodes.

7.2. Investigating Runtime Support For Dynamic Graphs

In comparison with static graphs, dynamic graphs have different workload characteristics. Hence, to efficiently support dynamic graph applications, different components of the

underlying runtime system (scheduler, communication layer, memory manager etc.) must be tailored to meet the need of the varying workloads.

For example, an important aspect of distributed computing is *fault tolerance* [44]. To develop fault-tolerant applications, some runtimes such as Charm++ [2] have implemented different techniques such as checkpointing etc, with the intention of being able to restart an application from the closest point of failure. On the other hand, we have designed our algorithms so that they can refine results as the algorithm progress. This label-correcting approach is similar in the spirit of self-stabilization [40]. Self-stabilization can be helpful in environment where graph algorithms are executed in a system that needs to be fault-tolerant, such as dynamic graphs. Certain updates to the graph structure can be critical to maintain a globally acceptable answer. Ordering of vertices is one of the techniques we rely upon for eliminating sub-optimal work. The task scheduler of a runtime system for dynamic graphs can use the same technique to prioritize updates. Instead of determining the best global update, an algorithm can decide which update to execute, based on local knowledge. As we have advocated before, we envision that, such technique of approximating globally best updates by only inspecting local information can be useful when there are too many updates to consider.

To handle fault tolerance, inserting checkpoints to take snapshots of the current state of the dynamic graphs can be done. In [80], a shared-memory dynamic graph library has been designed based on this idea. However, designing and implementing such idea in distributed setting would be challenging.

7.3. Supporting Multiple Algorithms to Run Simultaneously

Challenges to support concurrent execution of multiple algorithms for different graph problems include efficient support for termination detection. While it is possible to implement termination detection for such cases based on four-counter based algorithms, more states and information need to be stored, maintained and communicated across different compute nodes to properly detect global quiescence.

7.4. Graph-Machine Learning Pipeline

Another interesting aspect that can be explored is the complete end-to-end implementation of a library that supports machine-learning applications that can be modeled/represented with graphs, supporting asynchronous execution of graph algorithms and feeding the output of such representation to a distributed machine learning model.

As an example, let us consider graph2vec [89] application, an unsupervised approach to learn graph representation. Here, a graph is considered as a document where vertices and edges are viewed as words and connections between them respectively. graph2vec first generates rooted subgraphs around every vertex and then learns the embedding of the whole graph by associating a vector representation with each vertex in the next step. These two steps can be considered as part of the graph algorithm execution stage of the pipeline. Once the graph representation is learned, it can be used to classify graphs. The later stage can be considered as the machine learning stage of the pipeline.

CHAPTER 8

Conclusion

With the advent of exascale era, many challenge problems would necessitate rethinking algorithm design techniques to harness the available parallelism on heterogeneous platforms of multi-core and massively parallel processors with different hardware and networking capabilities. Keeping in mind these upcoming challenges, in this dissertation, we have taken a closer look at the interaction between distributed graph algorithms and the lower-level software stack, collectively referred to as the runtime system. Graph problems epitomize large, irregular applications of the future. Over the decades of development, High Performance Computing (HPC) platforms have been optimized for problems exhibiting good locality and regular memory access and communication patterns, benefiting from caching and high-bandwidth regular collective operations. Large-scale distributed graph algorithms present new challenges due to their irregular remote memory access pattern, and communication-bound execution characteristics. In particular, we observe that synchronization barriers in graph algorithms can cause significant bottleneck for scalability. Such observation has led us to design graph algorithms that avoid global and vertex-centric barriers to achieve better performance (Chapter 3). To design these algorithms, we have categorized graph operations in two main categories: monotonic updates and non-monotonic updates of vertex properties. Graph algorithms performing monotonic updates generally employ global synchronization barriers while algorithms with non-monotonic updates employ vertex-centric barriers to limit sub-optimal work propagation. We relax such constraints for both cases and demonstrate that, by incorporating algorithmic and runtime techniques, algorithms can solve graph problems faster by eliminating barriers. Elimination of barriers allows our algorithms to progress optimistically, refining the results as the algorithms progress. However, speculative execution

can generate sub-optimal work. To handle sub-optimal work explosion, our algorithms approximate global ordering by local thread-level ordering (priority) of tasks. In addition, we employ fine-grained task-based execution model of asynchronous many-task runtimes (AMTs) with unbounded-depth active messages and termination detection to enable asynchronous propagation of messages and independent task execution.

Our synchronization-avoiding graph algorithms rely on the underlying runtime for timely delivery of messages to the application and proper scheduling of tasks. This complex interaction between distributed graph algorithms and underlying runtime mandates proper runtime support for performance (Chapter 4). To this end, our runtime utilizes optimization techniques such as message coalescing and flow control transparently. Runtime techniques such as flow control and scheduling help effective interleaving of communication and computation. Such interleaving is necessary for timely delivery of work to thread-local private queues.

In addition, we have extensively investigated runtime scheduling policies for synchronization-avoiding distributed graph algorithms. We have demonstrated that (Chapter 5) incorporating application-level plug-in scheduler act as the complement of the default runtime scheduler. Plug-in schedulers can enforce different scheduling policies for graph algorithms based on domain expert's knowledge and can provide feedback to the runtime scheduler so as to decide whether to progress the network, throttle sending messages etc. Such decisions can be made based on the hints available to the application-layer.

In the final part of the dissertation, we have demonstrated that dynamically adapting runtime parameters based on workload characteristics can boost the performance of different categories of graph algorithms (Chapter 6). For this purpose, we have categorized algorithms in two broader categories: relaxed-synchronous and completely asynchronous. In particular, we have shown that adaptive message coalescing helps algorithms that executes in super-steps (relaxed-synchronous algorithms) to run faster by accelerating the last stage of each of the super-steps. Other techniques for designing graph algorithms that execute algorithms in super-steps such as Gather-Apply-Scatter (GAS) can also benefit

from adaptive message coalescing. The other technique, adaptive flow control, helps asynchronous graph algorithms to execute faster by regulating the back pressure. It is to be noted that our investigation of the scheduling policies has been done in the application-level while adaptivity of runtime parameters has been investigated in the runtime level.

Our holistic approach of designing and analyzing synchronization-avoiding graph algorithms in conjunction with runtime optimization techniques has opened several new directions of research that would be of interest to the community.

Bibliography

[1] Big Red II at Indiana University. http://rt.uits.iu.edu/ci/systems/BRII.php#info. Accessed: 2016-04-17.

[2] Charm++ checkpointing. http://charm.cs.illinois.edu/manuals/html/charm++/21.html, note = Accessed: 2017-08-21.

[3] Charm++ documentation. http://charm.cs.illinois.edu/manuals/html/charm++/10.html. Accessed: 2016-08-20.

[4] Dimacs inputs. http://www.dis.uniroma1.it/challenge9/. Accessed: 2016-05-31.

[5] GASNet Home Page. https://gasnet.lbl.gov/. Accessed: 2017-12-07.

[6] Graphlab git repository. https://github.com/jegonzal/PowerGraph. Accessed: December 2017.

[7] HPX-5. https://gitlab.crest.iu.edu/extreme/hpx. Accessed: 2016-05-25.

[8] Laboratory of web algorithmics. http://law.di.unimi.it/datasets.php. Accessed: December 2017.

[9] MPI Performance Topics. https://computing.llnl.gov/tutorials/mpi_performance. Accessed: 2016-04-10.

[10] MPI_Init_thread. http://www.mpich.org/static/docs/v3.1/www3/MPI_Init_thread.html. Accessed: 2016-04-10.

[11] JR Allwright, R Bordawekar, PD Coddington, K Dincer, and CL Martin. A comparison of parallel graph coloring algorithms. *SCCS-666*, pages 1–19, 1995.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore

architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[13] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[14] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. SC '12, pages 66:1–66:11, 2012.

[15] S. Beamer, A. Buluç, K. Asanović, and D. Patterson. Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. In *International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 1618–1627, May 2013.

[16] Scott Beamer, Krste Asanović, and David Patterson. Direction-Optimizing Breadth-First Search. *Scientific Programming*, 21(3-4):137–148, 2013.

[17] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 41:1–41:12. ACM, 2013.

[18] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy Sequential Maximal Independent Set and Matching Are Parallel on Average. In *Proc. 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 308–317. ACM, 2012.

[19] Raphaël Bleuse, Thierry Gautier, João V. F. Lima, Grégory Mounié, and Denis Trystram. Scheduling data flow program in xkaapi: A new affinity based algorithm for heterogeneous architectures. *CoRR*, abs/1402.6601, 2014.

[20] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, April 1979.

[21] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive programming of gpu clusters with ompss.

In *IPDPS*, pages 557–568, 2012.

[22] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[23] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.

[24] AydÄśn Buluç, Scott Beamer, Kamesh Madduri, Krste Asanović, and David Patterson. Distributed-Memory Breadth-First Search on Massive Graphs. In *Parallel Graph Algorithms, D. Bader (editor), CRC Press*. 2015. To appear.

[25] Ümit V Çatalyürek, John Feo, Assefaw H Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10):576–594, 2012.

[26] Venkatesan T. Chakaravarthy, Fabio Checconi, Fabrizio Petrini, and Yogish Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 889–901, Washington, DC, USA, 2014.

[27] Venkatesan T Chakaravarthy, Fabio Checconi, Fabrizio Petrini, and Yogish Sabharwal. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. In *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*, 2014.

[28] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199 – 222, 1969.

[29] F. Checconi and F. Petrini. Massive data analytics: The Graph 500 on IBM Blue Gene/Q. *IBM Journal of Research and Development*, 57(1/2):10:1–10:11, January 2013.

[30] Fabio Checconi and Fabrizio Petrini. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *Proc. 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 425–434. IEEE, 2014.

[31] Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. Breaking the speed and scalability

barriers for graph exploration on distributed-memory machines. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 13:1–13:12. IEEE, 2012. Cited by 0023.

[32] Malolan Chetlur, Nael Abu-Ghazaleh, Radharamanan Radhakrishnan, and Philip A Wilsey. Optimizing communication in time-warp simulators. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 64–71. IEEE, 1998.

[33] Malolan Chetlur, Girindra D. Sharma, Nael B. Abu-Ghazaleh, Umesh Kumar V. Rajasekaran, and Philip A. Wilsey. An active layer extension to mpi. In *PVM/MPI*, 1998.

[34] Claude A Christen and Stanley M Selkow. Some perfect coloring properties of graphs. *Journal of Combinatorial Theory, Series B*, 27(1):49 – 59, 1979.

[35] Thomas F Coleman and Jorge J Moré. Estimation of sparse jacobian matrices and graph coloring blems. *SIAM journal on Numerical Analysis*, 20(1):187–209, 1983.

[36] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A Parallelization of Dijkstra's Shortest Path Algorithm. In *Mathematical Foundations of Computer Science 1998*, pages 722–731. Springer, 1998.

[37] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 37–48, Berkeley, CA, USA, 2014. USENIX Association.

[38] Timothy A. Davis and Yifan Hu. The u. of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 2011.

[39] Edsger W Dijkstra. A Note on Two Problems in Connexion With Graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[40] Edsger W Dijkstra. Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective*, pages 41–46. Springer, 1982.

[41] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[42] Nicholas Edmonds, Jeremiah Willcock, and Andrew Lumsdaine. Expressing Graph Algorithms Using Generalized Active Messages. In *Proc. 27th International ACM Conference on International Conference on Supercomputing*, pages 283–292. ACM, 2013.

[43] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine. Single-Source Shortest Paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, November 2006.

[44] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[45] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[46] Thejaka Kanewala et. al. Families of graph algorithms: Sssp case study. In *European Conference on Parallel Processing*, pages 428–441. Springer, 2017.

[47] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. Adaptive asynchronous parallelization of graph algorithms. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1141–1156. ACM, 2018.

[48] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[49] Guoyao Feng, Xiao Meng, and Khaled Ammar. Distinger: A distributed graph data structure for massive dynamic graph processing. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, BIG DATA '15, pages 1814–1822, Washington, DC, USA, 2015. IEEE Computer Society.

[50] Jesun Sahariar Firoz, Marcin Zalewski, Martina Barnas, Thejaka Amila Kanewala, and Andrew Lumsdaine. Context matters: Distributed graph algorithms and runtime

systems. In *Platform For Advanced Scientific Computing (PASC)*, 2016.

[51] Jesun Sahariar Firoz, Marcin Zalewski, Andrew Lumsdaine, and Martina Barnas. Runtime scheduling policies for distributed graph algorithms. In *Proceedings of the 32st IEEE International Parallel and Distributed Processing Symposium*, Piscataway, NJ, USA, 2018. IEEE Press.

[52] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. ParalleX: A study of a new parallel computation model. In *International Parallel and Distributed Processing Symposium*, pages 1–6, March 2007.

[53] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-complete Problems. In *Proc. 6th Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, 1974.

[54] Assefaw H. Gebremedhin, Duc Nguyen, Md. Mostofa Ali Patwary, and Alex Pothen. Colpack: Software for graph coloring and related problems in scientific computing. *ACM Trans. Math. Softw.*, 40(1):1:1–1:31, October 2013.

[55] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.

[56] Graph500. http://www.graph500.org/. Accessed: 2016-05-31.

[57] O. Green and D. A. Bader. custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2016.

[58] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[59] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL Parallel Graph Library. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, number 7760 in LNCS, pages 46–60. Springer Berlin Heidelberg, 2013.

[60] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *23rd International Conference on Parallel Architectures and Compilation*, pages 27–38, NY, USA, 2014. ACM.

[61] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 27–38. ACM, 2014.

[62] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 166–177, New York, NY, USA, 2014. ACM.

[63] Muhammad Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. Unordered:A Comparison of Parallelism and Work-Efficiencyin Irregular Algorithms. *ACM SIGPLAN Notices*, 46(8):3–12, 2011.

[64] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric Xing. More Effective Distributed ML Via A Stale Synchronous Parallel Parameter Server. In *Advances in Neural Info. Processing Systems*, pages 1223–1231, 2013.

[65] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru. An adaptive parallel algorithm for computing connected components. *IEEE Transactions on Parallel and Distributed Systems*, 28(9), 2017.

[66] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, May 1993.

[67] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[68] Thejaka Kanewala, Marcin Zalewski, and Andrew Lumsdaine. Parallel asynchronous distributed-memory maximal independent set algorithm with work ordering. In

24th Annual International Conference on High Performance Computing, 2017.

[69] Ezra Kissel and Martin Swany. Photon: Remote Memory Access Middleware for High-Performance Runtime Systems. In *First Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware*, IPDRM, 2016.

[70] Milind Kulkarni and Keshav Pingali. Scheduling issues in optimistic parallelization. In *IEEE International Parallel and Distributed Processing Symposium, 2007*, pages 1–7. IEEE, 2007.

[71] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic Parallelism Requires Abstractions. In *ACM SIGPLAN Notices*, volume 42, pages 211–222. ACM, 2007.

[72] Kevin Lang. Fixing two weaknesses of the spectral method. In *Advances in Neural Information Processing Systems*, pages 715–722, 2006.

[73] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[74] Jure Leskovec and Andrej Krevl. Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[75] L. Lovász, M. Saks, and W. T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Math.*, 75(1-3):319–325, September 1989.

[76] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in The Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[77] M Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proc. 17th Annual ACM Symposium on Theory of Computing*, STOC '85, pages 1–10, New York, NY, USA, 1985. ACM.

[78] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[79] Miao Luo, Dhabaleswar K Panda, Khaled Z Ibrahim, and Costin Iancu. Congestion avoidance on manycore high performance computing systems. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 121–132. ACM, 2012.

[80] Peter Macko, Virendra J Marathe, Daniel Wyatt Margo, and Margo I Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the 31st IEEE International Conference on Data Engineering*. IEEE, 2015.

[81] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav Pingali. Dsmr: A parallel algorithm for single-source shortest path problem. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 32:1–32:14, New York, NY, USA, 2016. ACM.

[82] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proc. SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010.

[83] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

[84] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[85] Ulrich Meyer and Peter Sanders. Δ-stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[86] A. Morari, A. Tumeo, D. ChavarrÃ■a-Miranda, O. Villa, and M. Valero. Scaling irregular applications through data aggregation and software multithreading. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1126–1135, May 2014.

[87] MPI Forum. Message Passing Interface (MPI) Forum Home Page. `http://www.mpi-forum.org/` (Dec. 2017).

[88] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500 benchmark. *Cray User's Group*, 2010.

[89] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017.

[90] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical Report UW-CSE-14-02-01, Univ. Washington, 2014.

[91] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[92] Donald Nguyen and Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 333–344, New York, NY, USA, 2011. ACM.

[93] Thap Panitanarak and Kamesh Madduri. Performance Analysis of Single-source Shortest Path Algorithms on Distributed-memory System. In *Proc. Sixth SIAM Workshop on Combinatorial Scientific Computing*, page 60, 2014.

[94] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[95] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory. In *Proc. 27th IEEE International Symposium on Parallel and Distributed Processing*, Los Alamitos, CA, USA, 2013. IEEE.

[96] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 549–559. IEEE,

2014.

[97] CongDuc Pham and Carsten Albrecht. Optimizing message aggregation for parallel simulation on high performance clusters. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1999. Proceedings. 7th International Symposium on*, pages 76–83. IEEE, 1999.

[98] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The Tao of Parallelism in Algorithms. *ACM SIGPLAN Notices*, 46(6):12–25, 2011.

[99] Xinyu Que, Fabio Checconi, Fabrizio Petrini, Xing Liu, and Daniele Buono. Exploring network optimizations for large-scale graph analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 26:1–26:10, New York, NY, USA, 2015. ACM.

[100] Rsocket. Rsocket Home Page. `http://rsocket.io/` (Dec. 2017).

[101] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. Graph colouring as a challenge problem for dynamic graph processing on distributed systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 30:1–30:12, Piscataway, NJ, USA, 2016. IEEE Press.

[102] Yossi Shiloach and Uzi Vishkin. An o (logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1), 1982.

[103] Amitabh Sinha, Laxmikant Kalé, and Balkrishna Ramkumar. A dynamic and adaptive quiescence detection algorithm. *UIUC tech report*, 1993.

[104] G. Slota, S. Rajamanickam, and K. Madduri. BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In *28th International Parallel and Distributed Processing Symposium*, pages 550–559, May 2014.

[105] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In

Proc. 19th Annual International Symposium on Computer Architecture, pages 256–266. ACM, 1992.

[106] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[107] Dominic JA Welsh and Martin B Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[108] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. AM++: A Generalized Active Message Framework. In *Proce. 19th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 401–410. ACM, 2010.

[109] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: a programming model for highly parallel fine-grained data-driven computations. In *Proc. 16th ACM symposium on Principles and practice of parallel programming*, pages 305–306. ACM, 2011.

[110] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: Parallel programming for data-driven applications. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM.

[111] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *Proc. 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015.

[112] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 25–25, November 2005.

[113] Marcin Zalewski, Thejaka Amila Kanewala, Jesun Sahariar Firoz, and Andrew Lumsdaine. Distributed Control: Priority Scheduling for Single Source Shortest Paths

Without Synchronization. In *Proc. of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, pages 17–24. IEEE, 2014.

[114] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):2091–2100, 2014.

[115] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

Jesun Sahariar Firoz

jesunsahariar@gmail.com |
LinkedIn : https://www.linkedin.com/in/jesun-s-firoz/
Gitlab : https://gitlab.com/jesunsahariar
Google scholar : bit.ly/googlescholarJesun

Education

Indiana University, Bloomington, Indiana, USA
PhD, Computer Science, *August' 12 - Dec' 18*
Advisor: Dr. Andrew Lumsdaine
Dissertation Title: Synchronization-Avoiding Graph Algorithms and Runtime Aspects
M.S., Computer Science, *August' 12 - May' 14*

Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
M.S., Computer Science and Engineering, *October' 09 - June' 12*
B.S., Computer Science and Engineering, *December' 04 - August' 09*

Research Interests

Distributed graph algorithms, Distributed runtimes, Performance evaluation, Large-scale machine learning.

Selected Publications

Jesun Sahariar Firoz, Marcin Zalewski, Thejaka Kanewala and Andrew Lumsdaine. **Synchronization-Avoiding Graph Algorithms**. In 25th IEEE International Conference on High Performance Computing (HiPC), Bengaluru, India, December 2018.

Jesun Sahariar Firoz, Marcin Zalewski, Joshua Suetterlein and Andrew Lumsdaine. **Adaptive Runtime Features For Distributed Graph Algorithms**. In 25th IEEE International Conference on High Performance Computing (HiPC), Bengaluru, India, December 2018.

Jesun Sahariar Firoz, Marcin Zalewski, Martina Barnas, and Andrew Lumsdaine. **Runtime Scheduling Policies for Distributed Graph Algorithms**. In 32st IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 640-649, May 2018.

Jesun Sahariar Firoz, Marcin Zalewski,and Andrew Lumsdaine. **POSTER: A scalable distance-1 vertex coloring algorithm for power-law graphs**. In proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP, pp. 391-392, 2018 (extended version in submission).

Jesun Sahariar Firoz, Thejaka Kanewala, Marcin Zalewski, Martina Barnas, and Andrew Lumsdaine. **Context Matters: Distributed Graph Algorithms and Runtime Systems: A Case Study of Distributed Graph Traversals**. In proceedings of the Platform for Advanced Scientific Computing Conference (PASC), 2016.

Jesun Sahariar Firoz, Martina Barnas, Marcin Zalewski, and Andrew Lumsdaine. **The Value of Variance**. In 7th ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 287-295, 2016.

Jesun Sahariar Firoz, Marcin Zalewski, Martina Barnas, and Andrew Lumsdaine. **Comparison Of Single Source Shortest Path Algorithms on Two Recent Asynchronous Many-task Runtime Systems.** In proceedings of the 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS), pp. 674-681, 2015.

Jesun Sahariar Firoz, M Sohel Rahman, Tanay Kumar Saha. **Bee algorithms for solving DNA fragment assembly problem with noisy and noiseless data**. In proceedings of the 14th ACM annual conference on Genetic and evolutionary computation (GECCO), pp. 201-208, 2012.

Jesun Sahariar Firoz, Masud Hasan, Ashik Zinnat Khan, M Sohel Rahman. **The 1.375 approximation algorithm for sorting by transpositions can run in** $O(n \ \ log \ \ n)$ **time**. Journal of Computational Biology, 18(8), pp.1007-1011.

Research Experience

NIAC, Pacific Northwest National Lab, Seattle, WA
PhD Intern June 2017 to August 2018
- Worked on implementing distributed graph applications and an Open Community Runtime (OCR)-based runtime (ARTS) to efficiently execute fine-grained graph applications on custom-designed hardware for graph processing.

Center For Research In Extreme Scale Technologies, Indiana University
Research Assistant January 2014 to May 2017
- Worked under the supervision of **Prof. Dr. Andrew Lumsdaine** and **Dr. Marcin Zalewski** to investigate the interaction between asynchronous many-task (AMT) runtimes (HPX-5 and AM++) and graph algorithms. We have designed and evaluated distributed graph algorithms that can avoid global and vertex-centric synchronization barriers via optimistic parallelism. We have built **libPXGL**, a graph library, based on High Performance Parallex **(HPX-5)** runtime, an implementation of Parallex execution model. Additionally worked on **Parallel Boost Graph Library version 2**, which is built on top of AM++ runtime.

Computer Skills

Languages: C, C++, Java, Python, Matlab, Assembly (x86)
Parallel Programming: MPI, OpenMP, Pthreads, AM++, HPX-5.

Relevant Graduate Coursework

Distributed Computing, Parallel Architecture and Programming, Advanced Computer Architecture, Parallelism in Programming Language and Systems, Cloud Computing, (Advanced) Scientific Computing, Sublinear algorithms for Big Data, High performance Computing, Object Oriented Software Development, Machine learning, Data Mining, Deep Learning for Speech Processing, Advanced Natural Language Processing, Vision for Robotics, Introduction to Intelligent Systems.